USENIX

3rd USENIX Symposium on Internet Technologies and Systems

San Francisco, California March, 2001

conference

proceedings

3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)

San Francisco, California USA
March 26–28, 2001

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $24 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past USITS Proceedings

2nd USITS Symposium	1999	Boulder, Colorado	24/30
1st USITS Symposium	1997	Monterey, California	20/26

ISBN 1-880446-14-6

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)

March 26–28, 2001
San Francisco, California, USA

Symposium Organizers

3rd USENIX Symposium on Internet Technologies and Systems

March 26–28, 2001

San Francisco, California, USA

Proceedings of the

3rd USENIX Symposium on

Internet Technologies and Systems

(USITS '01)

Measurement and Analysis of a Streaming-Media Workload

Maureen Chesire, Alec Wolman, Geoffrey M. Voelker†, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington †*University of California, San Diego*

Abstract

The increasing availability of continuous-media data is provoking a significant change in Internet workloads. For example, video from news, sports, and entertainment sites, and audio from Internet broadcast radio, telephony, and peer-to-peer networks, are becoming commonplace. Compared with traditional Web workloads, multimedia objects can require significantly more storage and transmission bandwidth. As a result, performance optimizations such as streaming-media proxy caches and multicast delivery are attractive for minimizing the impact of streaming-media workloads on the Internet. However, because few studies of streaming-media workloads exist, the extent to which such mechanisms will improve performance is unclear.

This paper (1) presents and analyzes a client-based streaming-media workload generated by a large organization, (2) compares media workload characteristics to traditional Web-object workloads, and (3) explores the effectiveness of performance optimizations on streaming-media workloads. To perform the study, we collected traces of streaming-media sessions initiated by clients from a large university to servers in the Internet. In the week-long trace used for this paper, we monitored and analyzed RTSP sessions from 4,786 clients accessing 23,738 distinct streaming-media objects from 866 servers. Our analysis of this trace provides a detailed characterization of streaming-media for this workload.

1 Introduction

Today's Internet is increasingly used for transfer of continuous-media data, such as video from news, sports, and entertainment Web sites, and audio from Internet broadcast radio and telephony. As evidence, a large 1997 Web study from U.C. Berkeley [10] found no appreciable use of streaming media, but a study three years later at the University of Washington found that RealAudio and RealVideo had become a considerable component of Web-related traffic [30]. In addition, new peer-to-peer networks such as Napster have dramatically increased the use of digital audio over the Internet. A 2000 study of the IP traffic workload seen at the NASA Ames Internet Exchange found that traffic due to Napster rose from 2% to 4% over the course of 10 months [13], and a March 2000 study of Internet traffic at the University of Wisconsin-Madison found that 23% of its traffic was due to Napster [20].

Streaming-media content presents a number of new challenges to systems designers. First, compared to traditional Internet applications such as email and Web browsing, multimedia streams can require high data rates and consume significant bandwidth. Second, streaming data is often transmitted over UDP [14], placing responsibility for congestion control on the application or application-layer protocol. Third, the traffic generated tends to be bursty [14] and is highly sensitive to delay. Fourth, streaming-media objects require significantly more storage than traditional Web objects, potentially increasing the storage requirements of media servers and proxy caches, and motivating more complex cache replacement policies. Fifth, because media streams have long durations compared to the request/response nature of traditional Web traffic, multiple simultaneous requests to shared media objects introduce the opportunity for using multicast delivery techniques to reduce the network utilization for transmitting popular media objects. Unfortunately, despite these new characteristics and the challenges of a rapidly growing traffic component, few detailed studies of multimedia workloads exit.

This paper presents and analyzes a client-based streaming-media workload. To capture this workload, we extended an existing HTTP passive network monitor [30] to trace key events from multimedia sessions initiated inside the University of Washington to servers in the Internet. For this analysis, we use a week-long trace of RTSP sessions from 4,786 university clients to 23,738 distinct streaming-media objects from 866 servers in the Internet, which together consumed 56 GB of bandwidth.

The primary goal of our analysis is to characterize this streaming-media workload and compare it to well-studied HTTP Web workloads in terms of bandwidth utilization, server and object popularity, and sharing patterns. In particular, we wish to examine unique aspects of streaming-media workloads, such as session duration, session bit-rate, and the temporal locality and degree of overlap of multiple requests to the same media object. Finally, we wish to consider the effectiveness of performance optimizations, such as proxy caching and multicast delivery, on streaming-media workloads.

Our analysis shows that, for our trace, most streaming-media objects accessed by clients are encoded at low bit-rates ($<$ 56 Kb/s), are modest in size ($<$ 1 MB), and tend to be short in duration ($<$ 10 mins). However, a small percentage of the requests (3%) are responsible for almost half of the total bytes downloaded. We find that the distributions of client requests to multimedia servers and objects are somewhat less skewed towards the popular servers and objects than with traditional Web requests. We also find that the degree of multimedia object sharing is smaller than for traditional Web objects for the same client population. Shared multimedia objects do exhibit a high degree of temporal locality, with 20–40% of active sessions during peak loads sharing streams concurrently; this suggests that multicast delivery can potentially exploit the multimedia object sharing that exists.

The rest of the paper is organized as follows. In Section 2 we provide an overview of previous related work. Section 3 provides a high-level description of streaming-media protocols for background. Section 4 describes the trace collection methodology. Section 5 presents the basic workload characteristics, while Section 6 focuses on our cache simulation results. Section 7 presents our stream merging results. Finally, Section 8 concludes.

2 Related Work

While Web client workloads have been studied extensively [3, 5, 10, 8, 29], relatively little research has been done on multimedia traffic analysis. Acharya et al. [1] analyzed video files stored on Web servers to characterize non-streaming multimedia content on the Internet. Their study showed that these files had a median size of 1.1 MB and most of them contained short videos ($<$ 45 seconds). However, since their study was based upon a static analysis of stored content, it is unclear how that content is actually accessed by Web clients.

Mena et al. [14] analyzed streaming audio traffic using traces collected from a set of servers at Broadcast.com, a major Internet audio site. The focus of their study was on network-level workload characteristics, such as packet size distributions and other packet flow characteristics. From their analyses, they derive heuristics for identifying and simulating audio flows from Internet servers. Their study showed that most of the streaming-media traffic (60–80%) was transmitted over UDP, and most clients received audio streams at low bit-rates (16–20 Kb/s). The most striking difference between results obtained from their trace and our analysis is that most of their streaming sessions were long-lived; 75% of the sessions analyzed lasted longer than one hour. We attribute this difference to the fact that they studied server-based audio traces from a single site, while we study a client-based trace of both audio and video streams to a large number of Internet servers.

Van der Merwe et al. [15] extended the functionality of tcpdump (a popular packet monitoring utility) to include support for monitoring multimedia traffic. Although the primary focus of their work was building the multimedia monitoring utility, they also reported preliminary results from traces collected from WorldNet, AT&T's commercial IP network. As in [14], their study of over 3,000 RTSP flows also focused on network-level characteristics, such as packet length distributions and packet arrival times. In addition, they characterized the popularity of object accesses in the trace and similarly found that they matched a Zipf-like distribution. In contrast to our university client trace, the WorldNet workload peaks later in the evening and has relatively larger weekend workloads. We attribute this difference to the time-of-day usage patterns of the two different client populations; WorldNet users are not active until after work, while the university users are active during their work day.

There has been significant commercial activity recently (by companies such as FastForward Networks, Inktomi, and Akamai) on building caching and multicast infrastructure for the delivery of both on-demand and live multimedia content. However, little has been published about these efforts.

Our paper builds upon this previous work in a number of significant ways. First, we study a trace with an order of magnitude more sessions. Second, we focus on application-level characteristics of streaming-media workloads – such as session duration and sizes, server and object popularity, sharing patterns, and temporal locality – and compare and contrast those characteristics with those of non-streaming Web workloads. Lastly, we explore the potential benefits of performance optimizations such as proxy caching and multicast delivery on streaming-media workloads.

3 Streaming Media Background

This section defines a number of terms and concepts that are used throughout the paper. We use the term streaming media to refer to the transfer of live or stored multimedia data that the media player can render as soon as it is received (rather than waiting for the full download to complete before rendering begins). Although streaming techniques are typically used to transfer audio and video streams, they are sometimes used to deliver traditional media (such as streaming text or still images). A wide variety of streaming-media applications are in use today on the Internet. They employ a wide variety of protocols and algorithms that can generally be classified into five categories:

1. **Stream control protocols** enable users to interactively control the media stream, e.g., pausing, rewinding, forwarding or stopping stream playback. Examples include RTSP[26], PNA[21], MMS[16] and XDMA[31]. These protocols typically rely on TCP as the underlying transport protocol.

2. **Media packet protocols** support real-time data delivery and facilitate the synchronization of multiple streams. These protocols define how a media server encapsulates a media stream into data packets, and how a media player decodes the received data. Most media packet protocols rely on UDP to transport the packets. Examples include RDT[22], RTP[25], PNA[21], MMSU and MMST[16].

3. **Encoding formats** dictate how a digitized media stream is represented in a compact form suitable for streaming. Examples of encoding schemes commonly used include WMA[16], MP3[19], MPEG-2[18], RealAudio G2 and RealVideo G2 [23].

4. **Storage formats** define how encoded media streams are stored in "container" files, which hold one or more streams. Headers in the container files can be used to specify the properties of a stream such as the encoding bit-rate, the object duration, the media content type, and the object name. ASF[9] and RMFF[2] are examples of container file formats.

5. **Metafile formats** provide primitives that can be used to identify the components (URLs) in a media presentation and define their temporal and spatial attributes. SDP[11], SMIL[28] and ASX[17] are examples of metafile formats.

4 Methodology

We collected a continuous trace of RTSP traffic flowing across the border routers serving the University of Washington campus over a one week period between April 18th and April 25th, 2000. In addition to monitoring RTSP streams, the trace tool also maintained connection counts for other popular stream control protocols: PNA[21] used by Real Networks' servers, MMS[16] used by Microsoft Windows Media servers, and XDMA[31], used by Xing's StreamWorks servers. Our trace data was collected using a DEC Alpha workstation connected to four Ethernet switches at the University network border. The software used to monitor and log streaming-media packets is described in Section 4.2.

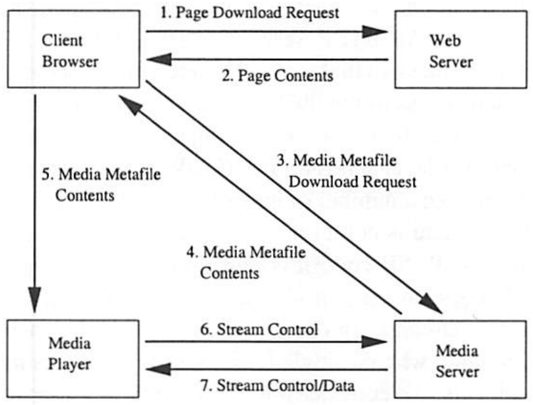


Figure 1: *Streaming media communication.*

4.1 Protocol Processing

Capturing streaming-media traffic is challenging because applications may use a variety of protocols. Moreover, while efforts have been made to develop common standardized protocols, many commercial applications continue to use proprietary protocols. Given the diversity of protocols, we decided to focus initially on the standardized and well documented RTSP protocol [26]. Widely used media players that support RTSP include Real Networks' RealPlayer and Apple's QuickTime Player. A high level description of the RTSP protocol is presented below.

4.1.1 RTSP Overview

The RTSP protocol is used by media players and streaming servers to control the transmission of media streams. RTSP is a request-response protocol that uses a MIME header format, similar to HTTP. Unlike HTTP, the RTSP protocol is stateful and requests may be initiated by either clients or servers. In practice, the RTSP control traffic is always sent over TCP. The media data is often sent over UDP, but it may also be interleaved with the control traffic on the same TCP connection. RTSP uses sequence numbers to match requests and responses. Media objects are identified by an RTSP URL, with the prefix "rtsp:".

Figure 1 illustrates a common streaming media usage scenario. First, a user downloads a Web page that contains a link to a media presentation. This link points to a metafile hosted by the media server. The Web browser then downloads the metafile that contains RTSP URLs for all the multimedia objects in the presentation (e.g., a music clip and streaming text associated with the audio). Next, the browser launches the appropriate media player and passes the metafile contents to the player. The media player in turn parses the metafile and initiates an RTSP connection to the media server.

Many of our analysis results will refer to an RTSP "session." An RTSP session is similar to an HTTP "GET" request, in that typically there will be one session for each access to the object. A session begins when the media player first accesses the object, and it ends when the media player sends a TEARDOWN message, though there may be a number of intervening PAUSE and PLAY events. There is not a one-to-one mapping between sessions and RTSP control connections; instead, the protocol relies on session identifiers to distinguish among different streams. In order to make our results easier to understand, when a single RTSP session accesses multiple objects, we consider it to be multiple sessions – one for each object.

4.2 Trace Collection and Analysis Software

We extended our existing HTTP passive network monitor [30] to support monitoring the RTSP streaming-media protocol. Our trace collection application has three main components: a packet capture and analysis module that extracts packet payloads and reconstructs TCP flows and media sessions; a protocol parsing module that parses RTSP headers; and a logging module that writes the parsed data to disk. After the traces are produced, we analyze the collected trace data in an off-line process. We now provide a brief overview of the operation of the trace collection software.

The packet capture module uses the Berkeley packet filter [12] to extract packets from the kernel and then performs TCP reassembly for each connection. Simple predicates are used on the first data bytes of each connection to classify TCP connections as RTSP control connections. This module also maintains a session state table that is used to map control messages to client streaming sessions; each table entry models the state machine for a client's multimedia session. In addition to maintaining the session table, this module provides support for: handling responses that arrive earlier than the corresponding requests due to asymmetric routing; merging control messages that cross packet boundaries; and extracting messages from connections transmitting control data interleaved with media stream data.

Data in the control connections is used to determine which UDP datagrams to look at. We record timing and size information about the UDP data transfers, but we do not attempt to process the contents of media stream packets because almost all commonly used encoding formats and packet protocols are proprietary.

The protocol parsing module extracts pertinent information from RTSP headers such as media stream object names, transport parameters and stream play ranges. All sensitive information extracted by the parser, such as IP addresses and URLs, is anonymized to protect user privacy. Finally, the logging module converts the data to a compact binary representation and then saves it to stable storage.

Attribute	Values
Connection counts	58808 (RTSP); 44878 (MMS); 35230 (PNA); 3930 (XDMA)
RTSP Servers	866 (External)
RTSP Total Bytes	56 GB (Continuous media)
RTSP Clients	4786 (UW clients)
RTSP Sessions	40070 (Continuous media)
RTSP Objects	23738 (Continuous media); 3760 (Other)

Table 1: Trace statistics.

5 Workload Characterization

This section analyzes the basic characteristics of our streaming-media workload; when appropriate we compare these characteristics to those of standard Web object workloads. The analysis ignores non-continuous media data (e.g., streaming text and still images). Since we were interested in the access patterns of the UW client population, we ignored sessions initiated by clients external to UW that accessed servers inside the campus network.

Table 1 summarizes the high-level characteristics of the trace. During this one-week period, 4,786 UW clients accessed 23,738 distinct RTSP objects from 866 servers, transferring 56 GB of streaming media data. Using the connection counts from Table 1, we estimate that RTSP accounts for approximately 40% of all streaming media usage by UW clients.

The detailed analyses in the following sections examine various attributes of the traffic workload, such as popularity distributions, object size distributions, sharing patterns, bandwidth utilization, and temporal locality. Overall, our analysis shows that:

- Most of the streaming data accessed by clients is transmitted at low bit-rates: 81% of the accesses are transmitted at a bit-rate less that 56 Kb/s.
- Most of the media streams accessed have a short duration ($<$ 10 minutes) and a modest size ($<$ 1 MB).
- A small percentage of the sessions (3%) are responsible for almost half of the bytes downloaded.
- The distribution of client requests to objects is Zipf-like, with an α parameter of 0.47.
- While clients do share streaming-media objects, the degree of object sharing is not as high as that observed in Web traffic traces [5, 30].

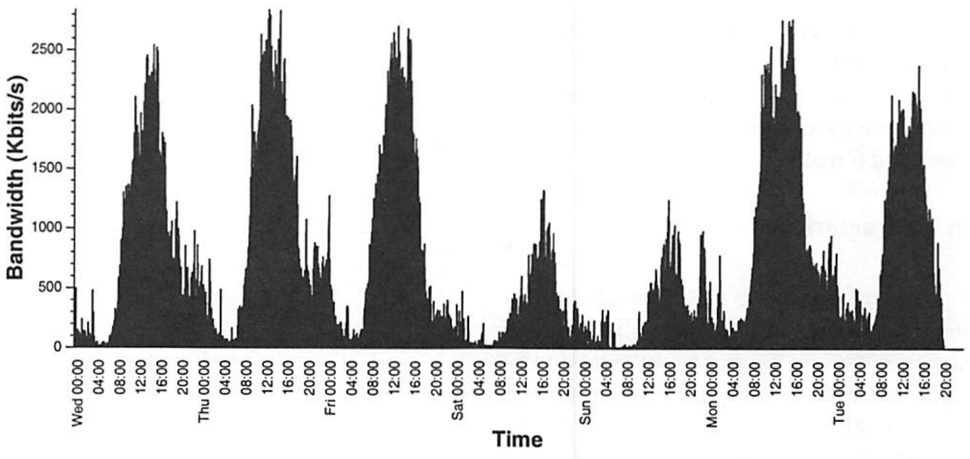
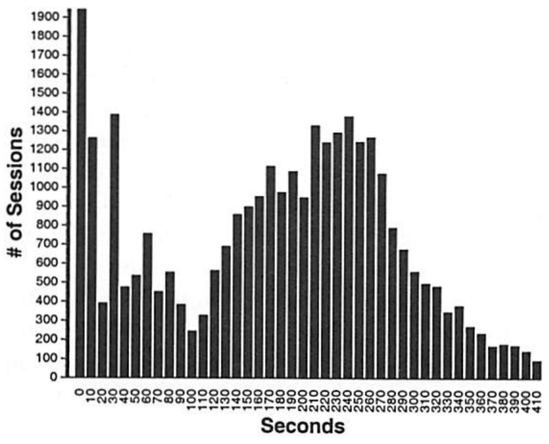


Figure 2: *Bandwidth utilization over time (in Kbits/sec).*



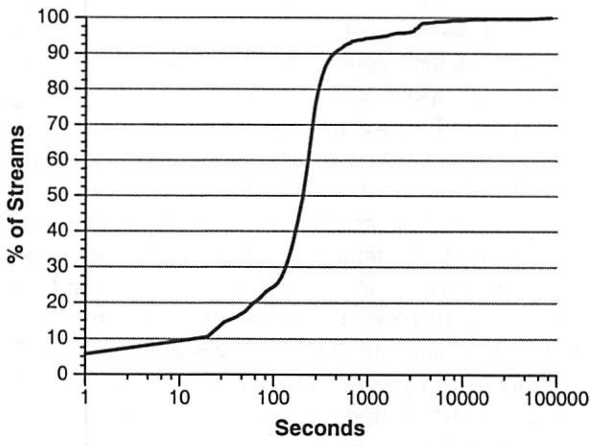


Figure 3: *Advertised stream length. (a) Normal and (b) CDF.*

- There is a high degree of temporal locality in client requests to repeatedly accessed objects.

5.1 Bandwidth Utilization

Figure 2 shows a time-series plot of the aggregate bandwidth consumed by clients streaming animations, audio, and video data. We see that the offered load on the network follows a diurnal cycle, with peaks generally between 11 AM and 4 PM. The volume of traffic is significantly lower during weekends; peak bandwidth over a five-minute period was 2.8 Mb/s during weekdays, compared to 1.3 Mb/s on weekends. We found that, on average, clients received streaming content at the rate of 66 Kb/s. This bit-rate is much lower than the capacity of the high-bandwidth links of the clients and the UW ISP links. We conclude from the prevalence of these low-bit-rate sessions that the sites that clients are accessing encode streaming content at modem bit-rates, the lowest common denominator.

5.2 Advertised Stream Length

In this section, we provide a detailed analysis of the advertised duration of continuous media streams referenced during the trace. Note that the advertised duration of a stream is different from the length of time that the client actually downloads the stream (e.g., if the user hits the stop button before the stream is finished). Since media servers generally do not advertise the duration of live streams, we limit our analysis to on-demand (stored) media streams. Sessions accessing these on-demand streams account for 85% of all sessions.

Figure 3a is a histogram of all streams lasting less than seven minutes, and Figure 3b plots the cumulative distribution of all stream lengths advertised by media servers. The peaks in the histogram in Figure 3a indicate that many streams are extremely short (less than a minute), but the most common stream lengths are between 2.5 and 4.5 minutes. These results suggest that clients have a stronger preference for viewing short multimedia streams. One important observation from Figure 3b is that the stream-length distribution has a long tail. Although the vast majority of the streams (93%) have a duration of less than 10 minutes, the remaining 7% of the objects have advertised lengths that range between 10 minutes and 6 weeks.

5.3 Session Characteristics

In this section, we examine two closely related properties of sessions: the amount of time that a client spends accessing a media stream, and the number of bytes downloaded during a stream access. In Figure 4 we present the relationship between the duration of a streaming-media session and the number of bytes transferred. In Figure 5 we look at the distinguishing characteristics between sessions accessing shared objects and sessions accessing unshared objects. Finally, in Figure 6 we compare the size and duration characteristics of sessions from clients in the campus modem pool to sessions from clients connected by high-speed department LANs.

A number of important trends emerge from these graphs. First, we see that client sessions tend to be short. From Figure 5a we see that 85% of all sessions (the solid black line) lasted less than 5 minutes, and the median session duration was 2.2 minutes. The bandwidth consumed by most sessions was also relatively modest. From Figure 5b we see that 84% of the sessions transferred less than 1 MB of data and only 5% accessed more than 5 MB. In terms of bytes downloaded, the median session size was 0.5 MB. Both the session duration and the session size distributions have long tails: 20 sessions accessed more than 100 MB of data each, while 57 sessions remained active for at least 6 hours, and one session was active for 4 days. Although the long-lived sessions (> 1 hour) account for only 3% of all client sessions, these sessions account for about half of the bandwidth consumed by the workload. From Figure 4 we see that these long sessions account for 47% of all bytes downloaded.

Most of the media objects accessed are downloaded at relatively low bit-rates despite the fact that most of the clients are connected by high-speed links. Using the raw data from Figure 4, we calculated that 81% of the streams are downloaded at bit-rates less than 56 Kb/s (the peak bandwidth supported by most modems today). In Figure 6, we separate all the trace sessions into those made from clients in the modem pool and those made from LAN clients. Although it does appear that the duration of modem sessions is shorter than the duration of LAN sessions (Figure 6a), the difference is not large. On the other hand, the difference in bytes downloaded between modem sessions and LAN sessions (Figure 6b) appears to be much more pronounced. For modem users, the median session size is just 97 KB, whereas for LAN users it is more than 500 KB.

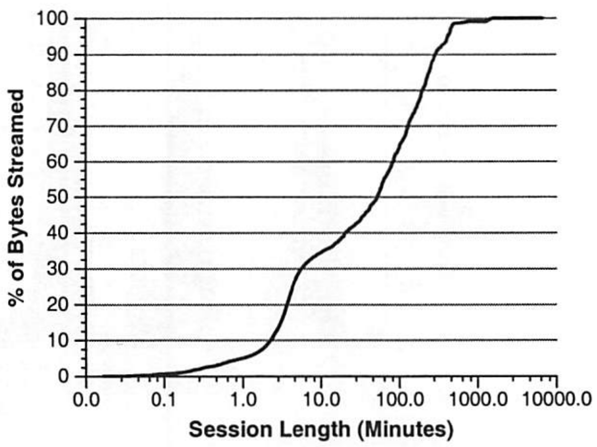


Figure 4: *Session duration vs. session size.*

Figures 5a and 5b also distinguish between accesses to shared objects (the dashed lines) and accesses to unshared objects (the grey lines). A shared object is one that is accessed by more than one client in the trace; an unshared object is accessed by only one client, although it may be accessed multiple times. Overall, sessions that request shared objects tend to be shorter than sessions accessing unshared objects. For example, 46% of shared sessions lasted less than one minute, compared with only 30% of the unshared sessions. Furthermore, we found that most of the sessions accessing shared objects transferred less data than unshared sessions. For example, 44% of shared sessions transferred less than 200 KB of data compared to only 24% of unshared sessions. However, Figure 5 shows that the situation changes for sessions on the tails of both curves, where the sessions accessing shared objects are somewhat longer and larger than sessions accessing unshared objects.

5.4 Server Popularity

In this section we examine the popularity of media servers and objects. Figure 7 plots (a) the cumulative distribution of continuous media objects across the 866 distinct servers referenced, as well as (b) the cumulative distribution of requests to these servers. These graphs show that client load is heavily skewed towards the popular servers. For example, 80% of the streaming-media sessions were served by the top 58 (7%) media servers, and 80% of the streaming-media objects originated from the 33 (4%) most popular servers. This skew to popular servers is *slightly less pronounced* than for requests to non-streaming Web objects. From a May 1999 trace of the same client population, 80% of the requests to non-streaming Web objects were served by the top 3% of Web servers [30].

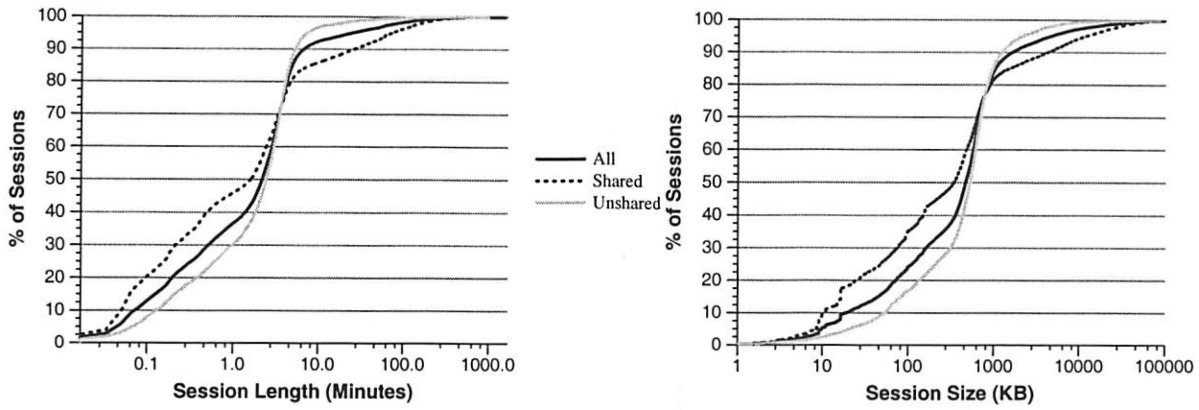


Figure 5: *Shared and unshared session characteristics. (a) Time and (b) Size.*

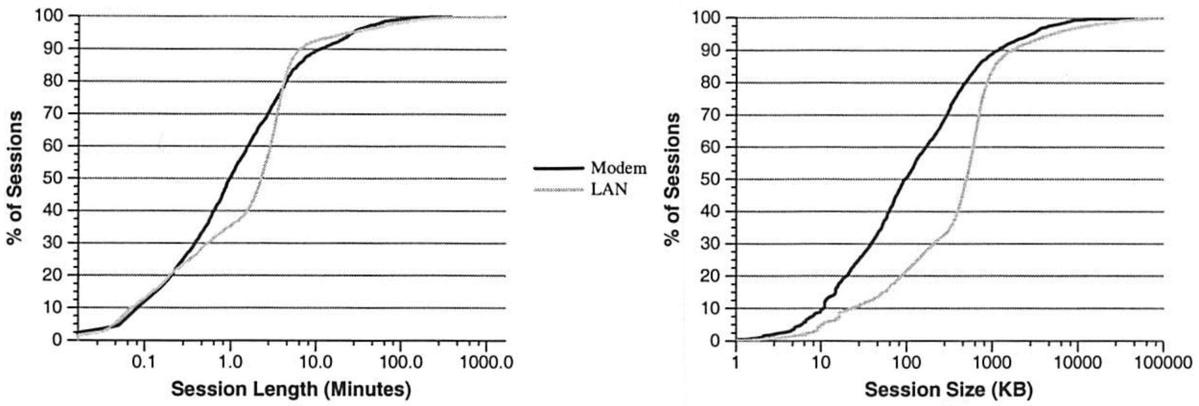


Figure 6: *Modem and LAN session characteristics. (a) Time and (b) Size.*

5.5 Object Popularity

One of the goals of our analysis was to understand how client requests were distributed over the set of multimedia streams accessed during the trace period. To determine this, we ranked multimedia objects by popularity (based on the number of accesses to each stream) and plotted the results on the log-scale graph shown in Figure 8. Our analysis found that of the 23,738 media objects referenced, 78% were accessed only once. Only 1% of the objects were accessed by ten or more sessions, and the 12 most popular objects were accessed more than 100 times each. From Figure 8, one can see that the popularity distribution fits a straight line fairly well, which implies that the distribution is Zipf-like [4]. Using the least squares method, we calculated the α parameter to be 0.47. In contrast, the α parameters reported in [4] for HTTP proxies ranged from 0.64 to 0.83. The implication is that accesses to streaming-media objects are somewhat less concentrated on the popular objects in comparison with previously reported Web object popularity distributions.

5.6 Sharing patterns

In this section we explore the sharing patterns of streaming-media objects among clients. We first examine the most popular objects to determine whether the repeated accesses come from a single client, or whether those popular objects are widely shared. In Figure 9, we compute the number of unique clients that access each of the 200 most popular streaming-media objects. This figure shows that the most popular streams are widely shared, and that as the popularity declines, so does the number of unique clients that access the stream.

Figure 10 presents per-object sharing statistics. Of the streaming-media objects requested, only 1.6% were accessed by five or more clients, while 84% were viewed by only one client. Only 16% of the objects were shared (i.e., accessed by two or more clients), yet requests for these shared objects account for 40% of all sessions recorded. From this data, we conclude that the shared objects are also more frequently accessed and can therefore benefit from caching. Note, however, that the degree of object sharing is low compared to the sharing rate for web documents [5, 30]. Consequently, multi-

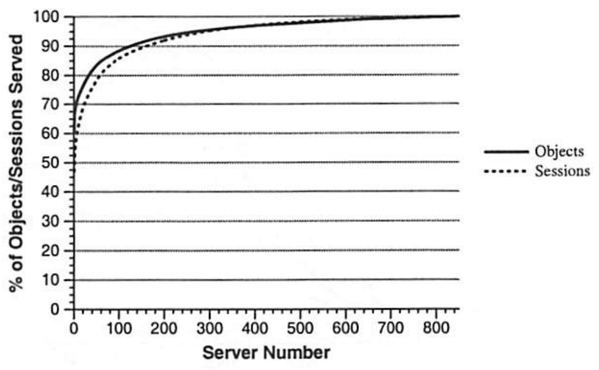


Figure 7: *Server popularity by object and session.*

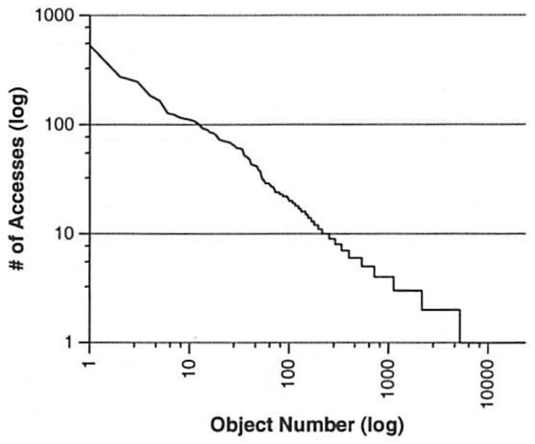


Figure 8: *Object popularity by number of sessions (note log scale).*

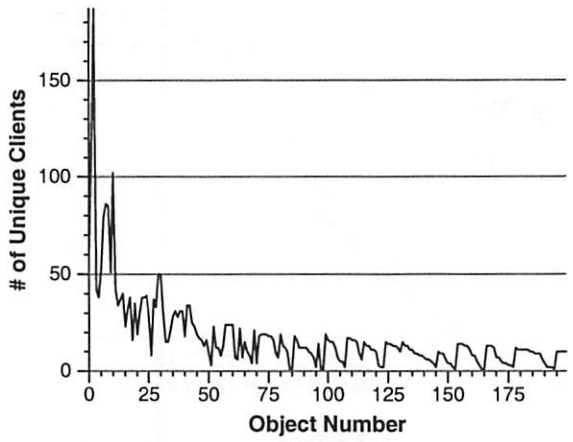


Figure 9: *Number of clients accessing popular objects.*

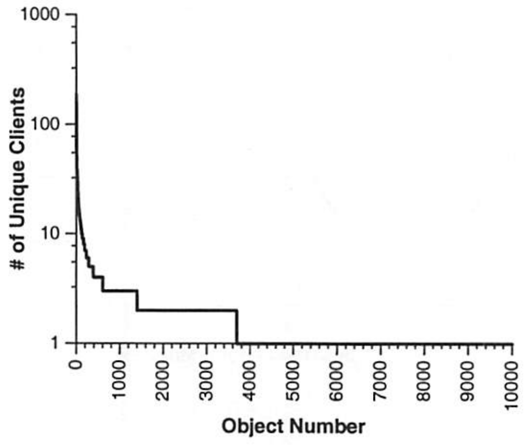


Figure 10: *Object sharing.*

media caching may not be as effective as Web caching in improving performance.

Figure 11 shows the overlap among accesses to the shared media objects by plotting the number of sessions that access unshared objects (black) and the number of sessions that access shared objects (grey) over time for the entire trace. During peak load periods between 11 AM and 4 PM (weekdays), we see that 20%–40% of the active sessions share streams concurrently. This temporal locality suggests that (1) caching will work best when it is needed the most (during peak loads), and that (2) multicast delivery has the opportunity to exploit temporal locality and considerably reduce network utilization.

6 Caching

Caching is an important performance optimization for standard Web objects. It has been used effectively to reduce average download latency, network utilization, and server load. Given the large sizes of streaming-media objects and the significant network bandwidth that they can consume, caching will also be important for improving the performance of streaming-media objects. In this section, we study the potential benefits of proxy caching for streaming-media objects. In particular, we determine cache hit rates and bandwidth savings for our workload, explore the tradeoff of cache storage and hit rate, and examine the sensitivity of hit rate to eviction timeouts.

We use a simulator to model a streaming media caching system for our analyses. The simulator caches the entire portion of any on-demand stream retrieved by a client, making the simplifying assumption that it is allowed to cache all stored media objects. For live streams, the simulator assumes that the cache can effectively merge multiple client accesses to the same live stream by caching a fixed-size sliding interval of bytes from that stream [7]. The simulator assumes unlimited cache capacity, and it uses a timeout-based cache eviction policy to expire cached objects. For Figures 12 through 14, an object is removed from the cache two hours after the end of the most recent access.

The results of the simulation are presented in the set of graphs below. Figure 12 is a time-series plot showing cache size growth over time, while Figure 13 shows

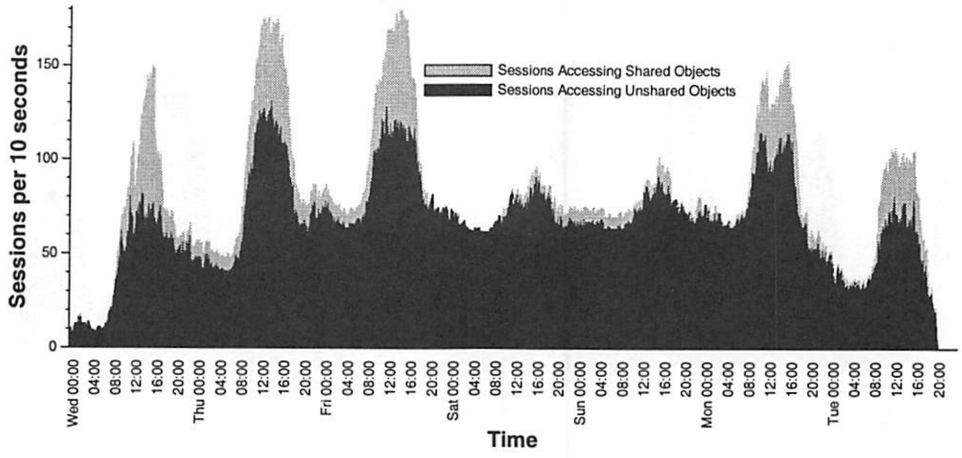


Figure 11: *Concurrent sharing over time.*

potential bandwidth savings due to caching. The total height of each bar in the stacked bar graph in Figure 14 reflects the total number of client accesses started within a one-hour time window. The lightest area of the graph shows the number of accesses that requested fully-cached objects; the medium-grey section represents the number of accesses that resulted in partial cache hits. Partial cache hits are recorded when a later request retrieves a larger portion of the media stream than was previously accessed. The height of the darkest part of the graph represents the number of accesses that resulted in cache misses.

Since streaming objects are comparatively large in size, the replacement policy for streaming proxy caches may be an important design decision. Many proposed designs for streaming proxy caches assume that multimedia streams are too large to be cached in their entirety [24, 27]. As a result, specialized caches are designed to cache only selected portions of a media stream; uncached portions of the stream have to be retrieved from the server (or neighboring caches) to satisfy client requests.

To determine the need for these complex caching strategies, we explored the sensitivity of hit rate to cache replacement eviction policies by varying the timeout for cached objects. Using the default two hour expiration of our simulator, we found that the simulated cache achieved an aggregate request hit rate of 24% (including partial cache hits) and a byte hit rate of 24% using less than 940 MB of cache storage. Because the required cache size is relatively small (when compared to the total 56 GB of data transferred), it appears that conventional caching techniques and short expiration times might be just as effective as specialized streaming media caching schemes for client populations similar to this.

Figure 15 plots request and byte hit rates as object eviction time is increased from 5 minutes to 7 days (the entire trace duration). Notice that reducing the caching window to 5 minutes still yields reasonably high request hit rates (20%). By keeping objects in the cache for only two hours after the last access, we achieve 90% of the maximum possible byte hit rate for this workload while saving significant storage overhead. From this data, we can infer that requests to streaming-media objects that are accessed at least twice have a high degree of temporal locality.

7 Stream Merging

Stream merging is a recently developed technique that uses multicast to reduce the bandwidth requirements for delivering on-demand streaming media. The details of stream merging are covered extensively in [6, 7]. In this section we provide a high level overview of stream merging to motivate our measurements.

Stream merging occurs when a new client requests a stream that is already in transmission. In this case, the server begins sending the client two streams simultaneously: (1) a "patch stream" starting at the beginning of the client's request, and (2) a multicast stream of the existing transmission in progress. The new client buffers the multicast stream while displaying the patch stream. When the patch stream is exhausted, the client displays the multicast stream from its buffer while it continues to receive and buffer the simultaneous multicast stream ahead of the display. At the merge point, only one stream, via multicast, is being transmitted to both clients. The cost of stream merging is that clients must be able to receive data faster than the required stream playback rate and must buffer the additional data.

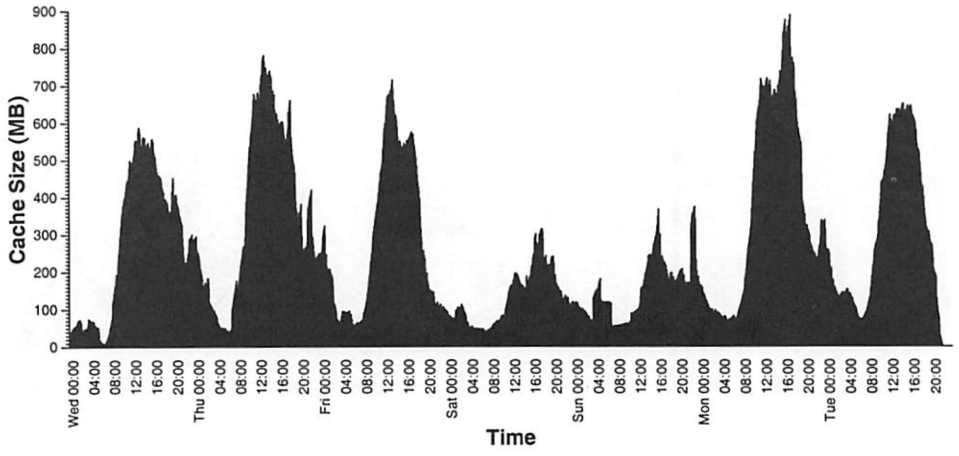


Figure 12: *Cache size growth over time.*

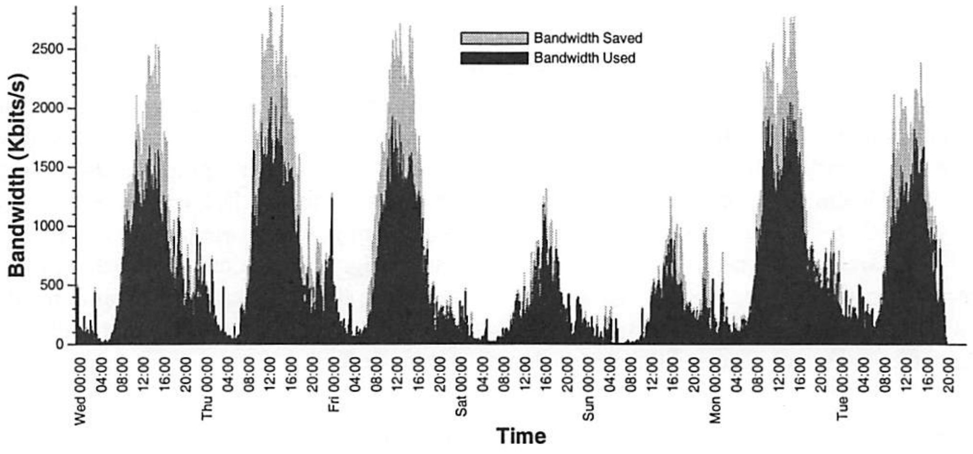


Figure 13: *Bandwidth saved over time due to caching.*

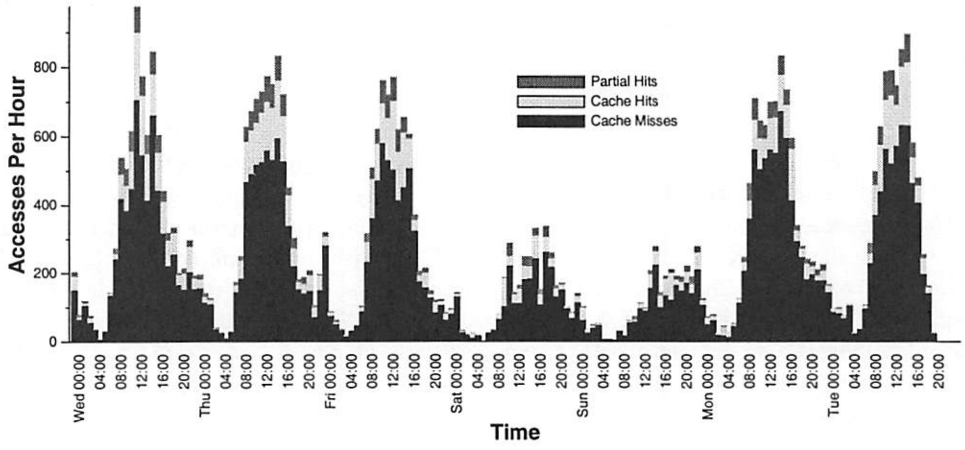


Figure 14: *Cache accesses: Hits, partial hits, and misses.*

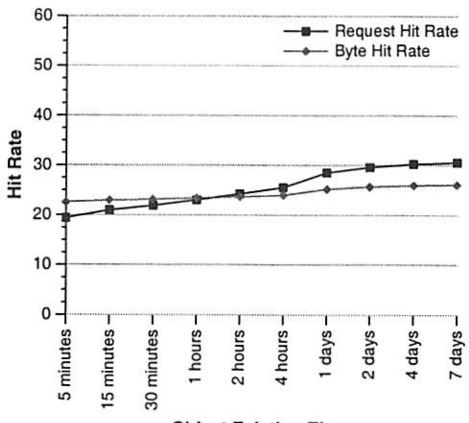


Figure 15: *Effect of eviction time on cache hit rates.*

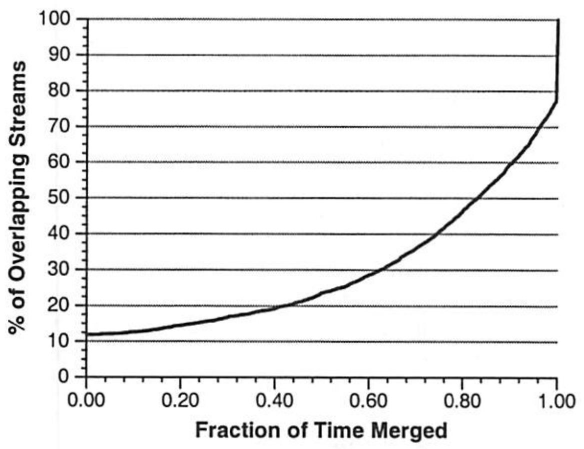


Figure 16: *Effectiveness of stream merging.*

To evaluate the effectiveness of stream merging for our workload, we consider consecutive overlapping accesses to each stream object in our trace and calculate the time it takes to reach the merge point based on [7]. Given the time of the merge point, we then calculate what percentage of the overlap period occurs after the merge point. This corresponds to the percentage of time that only one stream is being transmitted via multicast to both clients. The results of this analysis are shown as a cumulative distribution in Figure 16. Because this stream merging technique is only needed for on-demand streams, live streams are not included in Figure 16. This figure shows that stream merging is quite effective for this workload – for more than 50% of the overlapping stream accesses, shared multicast can be used for at least 80% of the overlap period. This result indicates strong temporal locality in our trace, which is consistent with our concurrent sharing and cache simulation results.

8 Conclusion

We have collected and analyzed a one-week trace of all RTSP client activity originating from a large university. In our analyses, we characterized our streaming multimedia workload and compared it to well-studied HTTP Web workloads in terms of bandwidth utilization, server and object popularity, and sharing patterns. In particular, we examined aspects unique to streaming-media workloads, such as session duration, session bit-rate, temporal locality, and the degree of overlap of multiple requests to the same media object. We also explored the effectiveness of performance optimizations, such as proxy caching and multicast delivery, on streaming-media workloads.

We have observed a number of properties (e.g., degree of object sharing and lower Zipf α parameter for the object-popularity distribution) indicating that multimedia workloads may benefit less from proxy caching, on average, than traditional Web workloads. On the other hand, we have found that multimedia workloads exhibit stronger temporal locality than we expected, especially during peak hours. This suggests that multicast and stream-merging techniques may prove to be useful for these workloads.

Our results are fundamentally based upon the workload that we captured and observed. It is clear that usage of streaming media in our environment is still relatively small, and our results could change as the use of streaming media becomes more prevalent. Our one-week trace contains only 40,000 sessions from fewer than 5,000 clients. A one-week trace of Web activity for the same population about a year earlier showed more than 22,000 active clients, and more than 80 million web requests during one week. Shifts in technology use, such as the widespread use of DSL and cable modems, will likely increase the use of streaming media and change underlying session characteristics. As the use of streaming media matures in the Internet environment, further client workload studies will be required to update our understanding of the impact of streaming-media data.

9 Acknowledgments

We would like to thank David Richardson, Terry Gray, Art Dong, and others at the UW Computing and Communications organization for supporting our effort to collect traces. We would also like to thank the USITS referees for their comments and suggestions. This research was supported in part by DARPA Grant F30602-97-2-0226, National Science Foundation Grant EIA-9870740, Compaq's Systems Research Center, and a Microsoft Graduate Research Fellowship.

References

[1] S. Acharya and B. Smith. An experiment to characterize videos stored on the web. In *Proc. of ACM/SPIE Multimedia Computing and Networking 1998*, January 1998.

[2] R. Agarwal, J. Ayars, B. Hefta-Gaub, and D. Stammen. RealMedia File Format. Internet Draft: draft-heftagaub-rmff-00.txt, March 1998.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *Proc. of IEEE Intl. Conference on Parallel and Distributed Information Systems '96*, Dec 1996.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of IEEE INFOCOM 1999*, March 1999.

[5] B. M. Duska, D. Marwood, and M. J. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, December 1997.

[6] D. Eager, M. Vernon, and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. In *Proc. of the 5th Int'l Workshop on Multimedia Information Systems*, October 1999.

[7] D. Eager, M. Vernon, and J. Zahorjan. Bandwidth skimming: A technique for cost-effective video-on-demand. In *Proc. of ACM/SPIE Multimedia Computing and Networking 2000*, January 2000.

[8] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proc. of IEEE INFOCOM 1999*, March 1999.

[9] E. Fleischman. Advanced Streaming Format (ASF) Specification. Internet-Draft: draft-fleischman-asf-01.txt, February 1998.

[10] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems*, December 1997.

[11] M. Handley and V. Jacobson. RFC 2327: SDP: Session Description Protocol, April 1998.

[12] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. of the Winter 1993 USENIX Technical Conference*, 1993.

[13] S. McCreary and K. Claffy. Trends in Wide Area IP Traffic Patterns: A View from Ames Internet Exchange. http://www.caida.org/outreach/pa pers/AIX0005/, May 2000.

[14] A. Mena and J. Heidemann. An empirical study of real audio traffic. In *Proc. of IEEE INFOCOM 2000*, March 2000.

[15] J. V. D. Merwe, R. Caceres, Y. hua Chu, and C. Sreenan. mmdump - a tool for monitoring multimedia usage on the internet. Technical Report 00.2.1, AT&T Labs, February 2000.

[16] Microsoft. Windows Media Development Center. http://msdn.microsoft.com/windowsmedia/.

[17] Microsoft. All About Windows Media Metafiles. http://msdn.microsoft.com/workshop/imedia/windows media/crcontent/asx.asp, April 2000.

[18] MPEG-2 Standard. ISO/IEC Document 13818-2. Generic Coding of Moving Pictures and Associated Audio Information, Part 2: Video, 1994.

[19] MPEG-2 Standard. ISO/IEC Document 13818-3. Generic Coding of Moving Pictures and Associated Audio Information, Part 3: Audio, 1994.

[20] D. Plonka. UW-Madison Napster Traffic Measurement. http://net.doit.wisc.edu/data/Napster, March 2000.

[21] RealNetworks. Firewall PNA Proxy Kit. http://www.service.real.com/firewall/pnaproxy.html.

[22] RealNetworks. RealNetworks Documentation Library. http://service.real.com/help/library/.

[23] RealNetworks. Realsystem production and authoring guides. http://service.real.com/help/library/encoders.html.

[24] R. Rejaie, M. Handley, H. Yu, and D. Estrin. Proxy caching mechanism for multimedia playback streams in the internet. In *Proc. of the Fourth Int. Web Caching Workshop*, March 1999.

[25] H. Schulzrinne, S. Casner, R. Fredrick, and V. Jacobson. RFC 1889: RTP: A Transport Protocol for Real-Time Applications, April 1996.

[26] H. Schulzrinne, A. Rao, and R. Lanphier. RFC 2326: Real Time Streaming Protocol (RTSP), April 1998.

[27] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *Proc. of IEEE INFOCOM 1999*, March 1999.

[28] W3C. Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. http://www.w3.org/TR/1998/REC-smil-19980615/, June 1998.

[29] C. E. Wills and M. Mikhailov. Towards a better understanding of web resources and server responses for improved caching. In *Proc. of the Eighth Int. World Wide Web Conference*, pages 153–165, May 1999.

[30] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of web-object sharing and caching. In *Proc. of the 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[31] Xingtech. Streamworks documentation. http://www.xingtech.com/support/docs/streamworks/.

Partial Prefetch for Faster Surfing in Composite Hypermedia

Javed I. Khan and Qingping Tao

Internetworking and Media Communications Research Laboratories
Department of Math & Computer Science
Kent State University
javed@kent.edu

Abstract

In this paper we present a prefetch technique, which incorporates a scheme similar to data streaming to minimize the response-lag. Unlike previous all or none techniques, we propose **partial prefetch** where the size of the lead segment is computed optimally so that only a minimum but sufficient amount of data is prefetched and buffered. The remaining segment is fetched if and only when the media is traversed. Thus, it delivers content without any increase in perceived response delay, and at the same time drastically minimizes unnecessary pre-load. The paper presents the scheme in the context of surfing in **composite multimedia** documents. It presents the technique and optimization scheme used for stream segmentation backed by analytical model and statistical simulation. We report remarkable increase in the responsiveness of web systems by a factor of 2-15 based on the specific situation.

Keywords: prefetch, streaming, multimedia.

1. Introduction

The success and failure of many web systems depend on the surfer's perception of the systems responsiveness. Caching and prefetching are the two principal techniques, known for improving responsiveness for systems that involve correlated data communication. Both the techniques have been extensively used in hardware systems to offset memory access latency. Caching techniques have been studied with much interest for the Internet. Also recently much focus has been shifted to prefetching. Some insight can be gained by a comparison between the two domains.

Despite the similarity, the caching in the Internet domain seems to be a harder problem. Hardware cache operates in a more predictable environment, where the parameters such as page size, and access times in the memory hierarchy are limited to few classes only. In comparison, the variability faced in the Internet is quite high. Apparently, the principle of locality is less conspicuous in the case of web. A number of recent studies, with various innovative caching schemes report caching efficacy in the range of 30-60% [4,7,10]. In comparison, a well-designed hardware cache can achieve a hit rate as high as 90% [1,18]. While several protocol issues are known to be involved in such performance degradation (such as validation requirement, and presence of Cookies and CGI scripts) but we also suspect one of the deeper reasons is the limited locality of reference. In the web, there is no short-term iterative construct such as 'while' of 'for' loops in the causal chain above. Consequently, it is not a complete surprise that the hit ratio is poorer here than in processor caches.

While, caching helps the case of repetitive references, prefetching reduces response lag for newer resources. It might be the case that prefetching may play much more important role in web than in hardware. A number of current studies, including ours, reported about another two times speedup with prefetching. Prefetching has also some additional complexity in the Web. Decision points mostly bifurcate the control flow tree in hardware due to the if-then construct. In contrast, the degree of branching in web is very high since there is no limit on the number of links in a page. In hardware quite often all the parallel branches are prefetched- and in some cases conditions can be pre-evaluated to determine the prefetch path. Neither is practical for web systems. We have also observed that prefetching often faces limitation due to excessive document transfer, which are never used later.

In this paper, we focus on this critical sub-problem-- ranking prefetchable date bytes. We discuss a technique, which provides optimum ranking of the prefetch nodes based on analytic considerations. In addition, we also show a technique where only a sub-part is fetched from the selected nodes. It cleverly uses the concept of *fragment streaming* to minimize the pre-load, without compromising the responsiveness gained by the prefetching in the first place. Each node is optimally divided into two parts-- the *lead* and the *stream* segments. During its operation, the system loads two parallel streams. In one, it fetches the stream segment of the current document, while in the other it prefetches the lead segment of candidate nodes.

Also we present the prefetch technique for an environment, which considers composite multimedia documents. An example is given is Fig-2 which shows a snapshot of a typical multimedia material from ABC News© website, serving combination of news summaries, images, voice integrated video clip as well as GIF animated banner advertisements from single link. Web pages with multiple embedded entities of various modalities such as applets, banners, audio, video clips, are becoming very common in power sites where responsiveness is critically valued. These are now typically composed of various media elements with varying playback requirements. In this paper, we also demonstrate how the size and mix of the lead segment can be computed optimally for such composite multimedia documents so that only a minimum but the right composition of individual media elements from these pages are prefetched and buffered. This problem has not been addressed before to such level.

2. Related Works

The Internet caching has been studied for quite some time [1,7,18]. It also has several implementations such as Harvest, Squid and CacheFlow [3,26]. The research in prefetching has gained momentum more recently [8,9,14]. In one of the pioneering studies, Kroeger et. al. demonstrated that with ample knowledge of future reference a combined caching and prefetching can reduce access latency as much as 60% [18,27]. The year after, Jacobson and Cao [14] proposed a prefetching method based on partial context matching for low bandwidth clients and proxies. Palpanas and Mendelzon [20] demonstrated that a k-order Markovian prediction engine similar to those used in image compression can improve response time by a factor of up to 2. Both these methods used variants of partial matching of context (past sequence of accessed references) for prediction of future web reference. These works suggested prefetching in-order of highest likely hood of access.

Pitkow and Pirolli [21] investigated various methods that have evolved to predict surfer's path from log traces such as session time, frequency of clicks, Levenshtein Distance analyses and compared the accuracy of various construction methods. This Markov model based study noted that although information is gained by studying longer paths, but conditional probability estimate, given the surf path, is more stable over time for shorter paths and can be estimated reliably with less data. Also of interest is the work by Cohen and Kaplan [4] who cited problems from bandwidth waste in prefetch, and as opposed to document prefetch suggested pre-staging only the communication session- such as pre-resolving DNS, pre-establishing of TCP connection and pre-warming

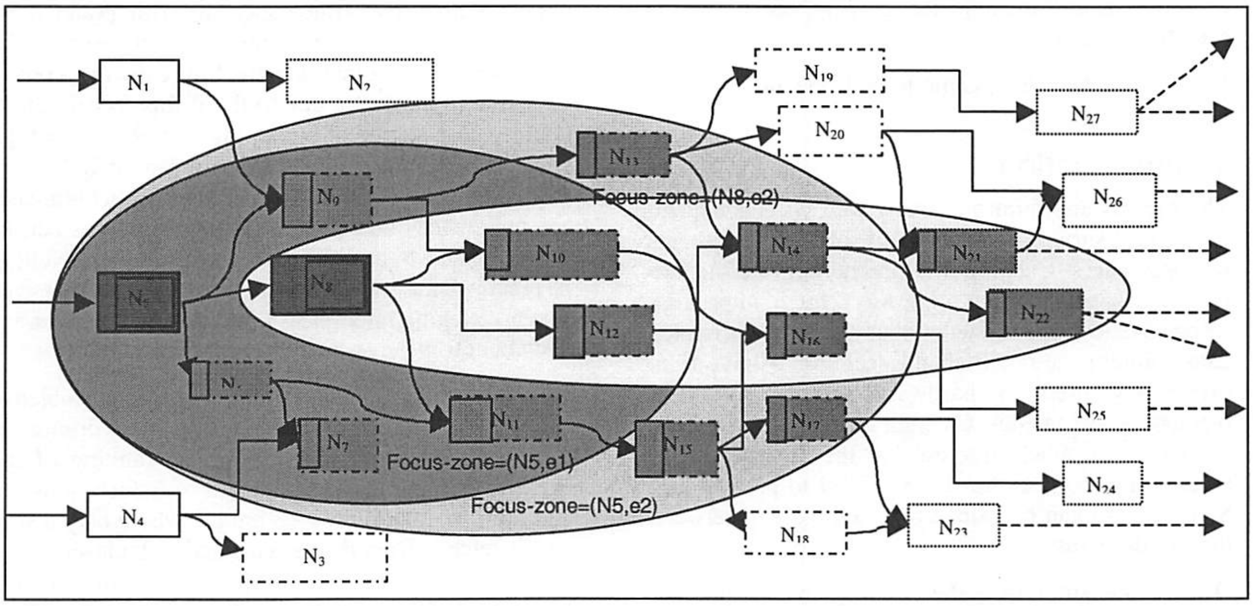


Fig-1: The hyperspace model used in the cache prefetching. The Focus-zones (N5,el) and (N5,e2) show the roaming sphere for two pruning thresholds both anchored at node N5. The priority algorithm ranks the nodes 5-12 or 5-17 based on the selected threshold. The prefetch mechanism loads the nodes accordingly, while the reader is reading N5. By the time reader completes reading N8 is ready for rendering. When reader moves to N8, the roaming sphere is updated. The dotted box shows the actual media size, while the solid box represents the lead segments those are actually pre-loaded.

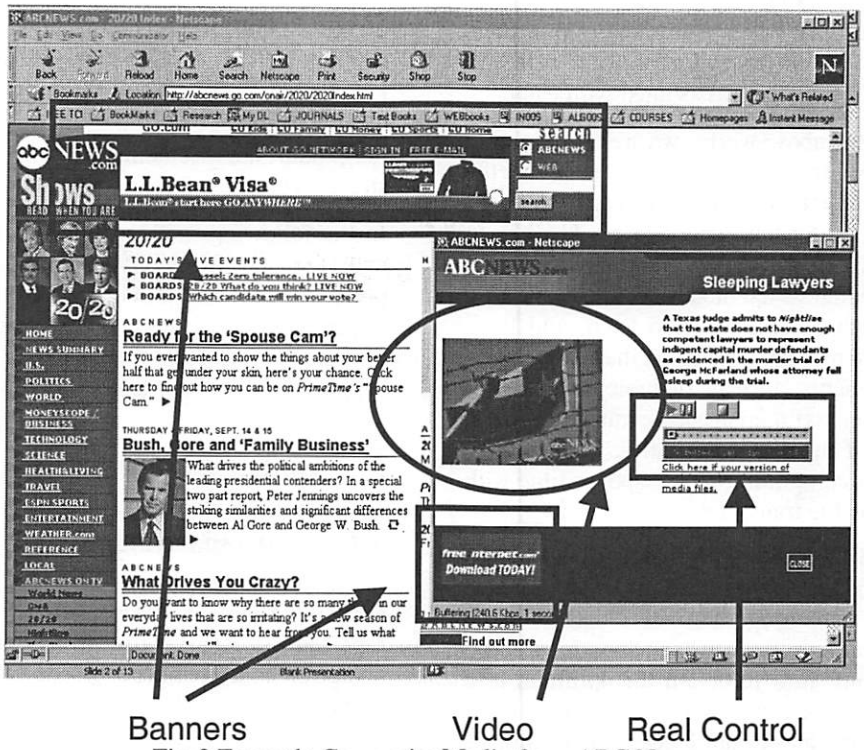


Fig-2 Example Composite Media from ABC News

Media Type	Files	KB
JPG images	5	18
GIF graphics	21	51
GIF animations		12
HTML	2	51
Video Stream	1	
Audio stream	1	
total	30	132

Table-1

	Comp.	Audio	Video
Codec:		16Kbps	RealVideo
		Voice	G2
minimum (Kbps)	86.8	~9	~77
maximum(Kbps)	123.4	~20	~102
Encoded/Clip BW	104.6	16	88.6
Average (Kbps):	121.7	18.7	92.8
Duration(min:sec)	3:12	3:12	3:12

Table-2

Table 1 and 2 Static and Dynamic Properties of the Media

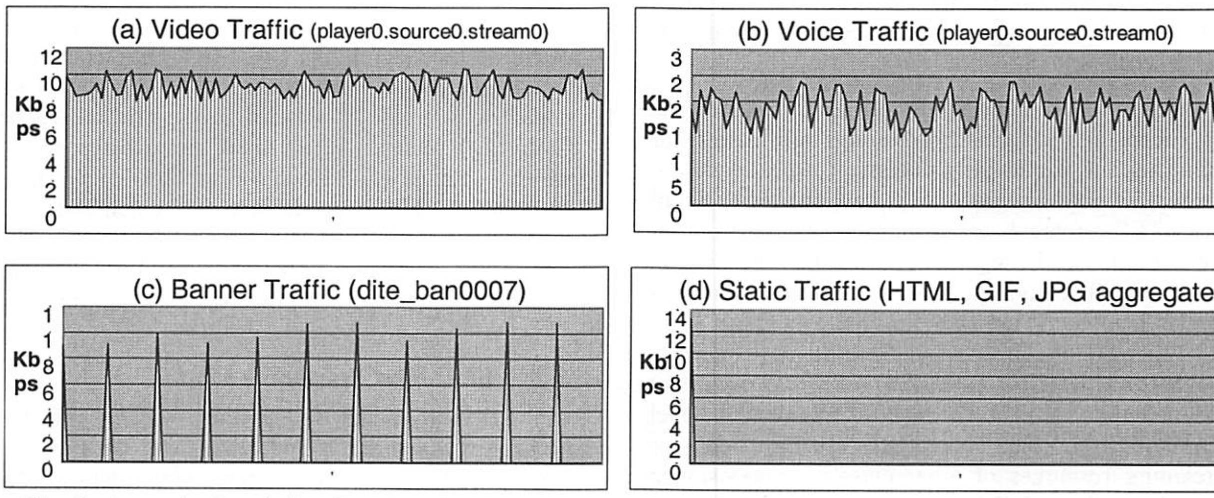


Fig- 3 shows the Rendering Rate Profiles respectively for the (a) Video (b) voice (c) Banner* (d) Graphics and Text and (d) composite media for the ABC News Nightline © Presentation.

by sending dummy HTTP HEAD request. RealPlayer (release 7) already pre-stages streaming connections linked from a page by pre-extracting and readying individual codec information associated with each.

In the context of the above works, we are investigating the issue of ranking-- particularly in view of the inevitability of prediction error. In a recent work we have shown ranking for low capacity links [16,17]. In this paper we extend the study in further depth—In addition to showing which nodes to fetch, we also analyze how much of each node to fetch, and for a composite document what should be the composition of the fetch segments. Instead of prefetching nodes simply in order of maximum probable transition paths, as used in most of the previous studies, we propose a ranking order which optimizes the response time with respect to all probable transition paths.

In this paper we further propose that instead of full documents we should only preload an estimated lead segment. The remaining can be loaded as a background streaming only when they have been requested. We show how to obtain the optimum lead segment that does not compromise the loading time-but reduces the amount of data loaded for documents that are never fetched.

The proposed method is based on linked transition probability and does not make any assumption about their means of estimation. It can be used with most of the methods proposed. Also to demonstrate its potential feasibility, in this paper we have sketched potential implementation technique only for the non-trivial steps. The actual deployment of prefetch system, including ours will require a much more detailed design work at the very protocol level. However, discussions about these issues cannot be accommodated in this paper. Rather, below we cite some additional interesting research, which we think will provide valuable insight about these aspects closely related to our work.

In related work, Duchamp [8] has discussed methods for clients and servers to exchange information for gathering and exchanging document usage statistics. Gruber, Rexford and Basso [12] have studied in detail the RTSP protocol extensions to support partial caching of lead segments of large multimedia documents. The concept of streaming for continuous media has been applied for quite a few years [15,22,23,24,25]. Grosso and Veillard have recently proposed XML extensions for document fragmentation [13]. Such fragment level communication can potentially be applied for streaming resources of other classes, as envisioned here. Crovella and Bradford [5] studied yet another advantage of prefetching- the reduction of burstiness of network traffic. Bangla et. al. proposed prefetching of only modified cached content to reduce cache communication in their proposal of 'optimistic-delta' [2]. Jung, Lee and Chon [15] reported error resilience of TCP prefetch-based long multimedia sessions on a long haul noisy channel as opposed to live UDP streaming.

In the following sections, we first present the model and the analytical considerations supporting the prefetch scheme. Section 4 then outlines key implementation issues. Finally, section 5 presents the performance of the scheme under various workload supported by rigorous statistical simulation.

3. Analysis

3.1 Hyperspace Model

First, we describe the model of the *hyperspace*. Fig-1 shows the model. Each *node* in the hyper-graph represents a web entity -- in the general case a composite multimedia document. Nodes are connected as per the embedded hyperlinks. The reader moves through a sequence of nodes in this hyperspace called *anchor sequence*. When a client (reader) program is active, the idea is to track the *anchor nodes*, monitor its neighboring regions, and prefetch selected parts from a subset of these nodes in client's *prefetch cache* with high likely-hood of traversal based on some optimization objective.

$H(V_H,E_H)$ denotes the entire hyperspace. $h(V_h,E_h)$ denotes the *visible sub-graph* of H about which the system has information. For tractability, it is further pruned before each optimization phase. We call this final sub-graph *roaming- sphere* and will denote it by $G(V_G,E_G)$. Although the node and link information of $h(V_h,E_h)$ or $G(V_G,E_G)$ is assumed available, but their content, however may not be resident in the prefetch cache at the beginning. Only $G(V_G,E_G)$ is used to determine the pre-loading schedule. A subset of its' nodes is eventually preloaded in the prefetch cache. $C(V_C, E_C)$ represents this final part of hyperspace, which is finally resident in the prefetch cache. Each node in $h(V_h,E_h)$ has node 'statistics' (such as size or load time). Also, each link in it has a transition probability p(i,j) associated with it. Links are bi-directional and transition probabilities are asymmetric.

3.2 Streamed Transport Model

Our transport model divides the transport into two probable phases. Each node thus has two transport parts-- the *lead segment* and the *stream segment*. The available bandwidth is correspondingly separated into two sub-channels; *feed channel* for loading the

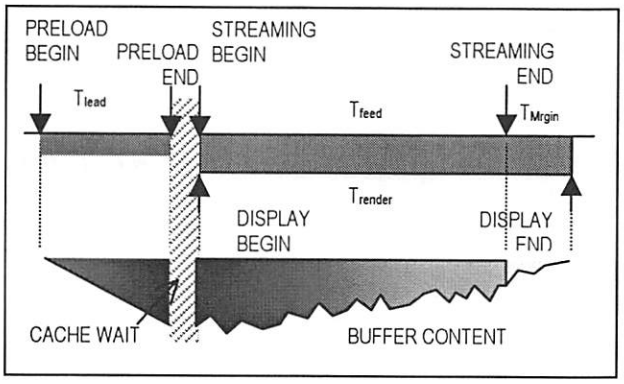


Fig-2 Streaming event model used for analysis. The top diagram shows event sequence and various time quantities and the bottom diagram shows the buffer occupancy under typical operation.

streaming segment of the current anchor, the *lead channel* to proactively load the lead segments of resources from the focus zone. Fig-4 shows the event sequence and an hypothetical buffer fullness for the transport model. The jaggedness of buffer fullness is due to the non-uniform nature of data consumption by the surfer. We assume that D_{total} is the size of an elementary resource, D_{lead} is the bytes in lead segment and D_{feed} is the bytes to be streamed. We use β to denote the ratio of total bandwidth to that allocated to the lead sub-channel. In Fig-1, the dotted box represents the media, while the solid head indicates the lead segments.

3.3 Composite Document Model

In this paper, we provide the entire technique in the context of composite document. A composite document has multiple embedded components with various rendering modalities. For example, Fig-2 shows a typical composite document with several embedded static as well as continuous media components (Tables -1 and 2 respectively provide a summary of the static and continuous media embedded here). To model such composite media we use a set of functions called *rendering rate profiles*-- one for each element. These profiles characterize the elements as a data rate function along the presentation time-line of the composite document. Note a profile is intrinsic to the document and is not dependent on the communication.

Given a composite document node N with embedded media elements n^i, we denote individual element's profile as the function $f^i(t)$, where superscript i is the element index. We use f(t) to denote the general variable rate profile. We also introduce the following three common profiles -- (a) constant rate (CBR) media, (b) impulse media, and (3) impulse series media.

Most CBR or audio and video plays generally are constant rate media. Parent HTML page, JAVA applets, or embedded foreground/background images generally contribute to the lead segment and is considered impulse media. A dynamic text or banner image requires cycles of bursts for presentation. Similar is the case with any large document (PDF, ps etc.) which are viewed page by page. Additional examples of this class of documents are stock ticks and charts, pushed banners, GIF animations as well as flash and slide shows. We use equation set-1 to model the profiles of these entities. Fig-5 sketches these functions.

$$\begin{aligned} f_c(t) &= R^i{}_c \\ f_I(t) &= H^i_I \\ f_S(t) &= \sum_{n=0} h^i_S \delta(t - nt^i_0) \end{aligned} \qquad ...(1)$$

The document's *composite rate profile* is the sum of individual functions characterized by the coefficients R, H and h.

Given the above analytical model the problem that we will address in the remaining part of this paper is the ranking analysis at three levels. Given $G(V_G,E_G)$-- a hyper linked space of composite documents, and the associated node statistics (transition frequency *p(i,j)*, and node profiles f(t)) *we show* how to obtain the optimum prefetch sequence that will generate minimum expected response lag. Further, we will also determine the lead vs. feed segment sizes of the composite documents that will minimize the lag but with minimum waste of overall bandwidth. And finally, for the chosen composite documents, we also show how much of which components of the composite documents have to be brought in and when, for both its lead and streamed segments -- i.e. the *segment transport schedule* for the optimum performance.

3.4 Prefetch Node Ranking

The first question we address is what is the best prefetching sequence of the nodes that will minimize the expected cumulative *read-time lag* for a given network bandwidth? Below we present the results in the form of a series of proofs that ultimately will lead to the ranking algorithm. Proofs of these theorems are given in [16].

First we define the optimization criterion. In a hyper-graph G, Lets $U=(a_1,a_2,a_3....a_l)$, where $u_i \in G$, is the *anchor sequence*-- the sequence of nodes followed by a user. Let's Γ is the *loading sequence* in which the nodes are loaded in the cache (Clearly, U $\subseteq$ Γ$\subseteq$ {nodes in G}). Let p_i is the estimated probability that a user traverses a node n_i in roaming sphere G, and $T_{L,i}$ is the time the node a_i is fetched and $T_{P,i}$ is the time spent by the user in that node. Thus, we define an overall penalty function-- the expected cumulative *read-time lag*:

$$T(\Gamma \mid U) = \sum_{i}^{U} p_i \max\{[T_{L,i} - (T_{L,i-1} + T_{P,i-1})],0\} \qquad ...(2)$$

The objective is to find the loading sequence Γ that will minimize the expected penalty E{T(Γ|U)}. It is important to note that this function optimizes with respect to all probable transitions of U, weighted by their transition probability. Given the above optimization criterion, it can be shown that:

***Theorem-1 (Branch Decision)**: Let $A=n_c$ is the current anchor point with direct transition paths to a set of candidate nodes n_1, n_2, n_3,.. n_n, such that T_i is the estimated loading times of node n_i, and $Pr[a_{n+1}=n_i \mid a_n=A]$ is the conditional link transition probability, then the average delay is minimum if the links are prefetched in-order of the highest priority Q_i, where:*

$$Q_i = \frac{\Pr[a_{n+1} = n_i \mid a_n = A]}{T_i} \qquad ...(3a)$$

This theorem states that at a simple branch point (roaming sphere of depth=1) immediately linked nodes should be prefetched in order of the conditional transition probability but in inverse order of their estimated load time.

The following theorem provides the relative priority between two nodes in a tree at different depths, which are not necessarily along the same path.

***Theorem-2 (Tree Decision)**: If the sequence {n_1, n_2, n_3,...n_d} are the nodes in the path in a tree from the current anchor $A=n_0$ to a candidate node n_d, at depth d and T_d is it's estimated loading time, then the priority Q_d is given by the product of the conditional transition probabilities along the path such that:*

$$Q_d = \frac{\prod_{i=c}^{d} \Pr[a_{i+1} = n_{i+1} \mid a_i = n_i]}{T_d} \qquad ...(3b)$$

A corollary of this second theorem provides the relative priority of the nodes along a sequence. The result is quite intuitive.

***Corollary-2.1 (Sequence Decision)**: If two nodes are in a sequence, then the preceding node has to be loaded first.*

Unfortunately, the relative priority cannot be determined for a general graph for the optimization criterion defined by equation (2). However, under a slightly modified definition which ignores the credit due to cumulative read times along the paths ($T_{p,i-1}=0$ for all i in equation-2) it can be shown:

***Theorem-3 (Graph Decision)**: For general graph $G(V_G,E_G)$ the node priority can be determined by computing order-n Markov state probability p_i. For a node n_i, with the estimated loading time T_i the priority function can be computed as:*

$$Q_i = \frac{p_i}{T_i} \qquad ...(3c)$$

The above results indeed provide a close form solution to the node-ranking problem. Let's consider C is the current set of nodes in the candidate node queue. The sequence corollary ranks the nodes those are in a path. In a tree-structured roaming-graph, they effectively prunes the set C only among the immediate next nodes of the anchor thus, if we are at node 0, the candidate set can be initialized at $C_0 = \{children\ of\ n_o\}$, The Branch Decision theorem then ranks the nodes within this set. Let the winner be n_{winner}. Then, when this is prefetched, it exposes its children nodes-- a new set of nodes, which now also becomes candidates for prefetch. Thus, the candidate set is updated to $C_i = C_{i-1} - \{n_{winner}\} + \{children\ of\ n_{winner}\}$. The Tree Decision theorem then provides the priroty of these new nodes compared to those which are already in the queue and thus completes the ranking. Finally, theorem-3 can be used in a graph for nodes which are child to multiple parents in C.

3.5 Critical Composite Lead Mass

The next question we address is how much of which node should we prefetch? This will also determine the pre-load time to be used in (2). Let us assume that the margin time=0. We use the constraint that the reading or rendering time should be at least equal to the streaming time for all components. First, we solve the simple case with only continuous media:

Continuous Media Case: Consequently, given a consumption rate R^i_{render} for the media type=*i* the amount of data that has to be prefetched is:

$$D^i_{lead} \geq D^i_{total}\left(1 - \frac{B^i_{feed}}{R^i_{render}}\right) = D^i_{total}\left(1 - \frac{\beta^i \cdot B_{channel}}{R^i_{render}}\right) \qquad .(4)$$

Only the D_{lead} amount of data should be pre-loaded for mininum delay. It provides a lower bound and we call it *critical lead mass*. Corresponding pre-load time for the media is given by:

$$T_{lead} = \frac{\sum_i D^i{}_{lead}}{B_{preload}} \qquad ...(5)$$

General case: The *critical lead mass* for a composite media can be determined by piece-wise integration of the combined rate function performed over a set of intervals defined as *negative zero crossing* (NZC) points where:

$$\sum f(t_i) - B_{feed} = 0 \text{ , and } \frac{d}{dt}\sum f(t_i) \leq 0 \cdot \qquad ...(6a)$$

Fig-6 explains the segments where t_1, t_2 and t_3 are three such points dividing the presentation time T into four segments. The size of the required minimum lead segment is then given by the maximum of the piecewise integrals evaluated in 0-t_i:

$$D_{lead} \geq \max_{j=1}^{N}\left(\int_{\varepsilon}^{t_j+\varepsilon} \sum_i f^i(t)dt - B_{feed} \cdot t_j \right) \qquad ...(6b)$$

For zero delay presentation, the system must preload equal or more. Note, if a segment sum negative in jth interval, than it can reduce the piecewise lead segment of (j+1)th interval, however, the reverse is not true. The quantity ε is a small positive interval for ensuring inclusion of impulses in preceding intervals. Consequently, we look for maximum positive growth in the rate difference function (difference between the *composite rate profile function* and the *feed channel bandwidth),* however, this can be evaluated only by integrating at NZC points. For example, in the last segment of Fig-5(a) Since, the negative area A is larger than the positive area B , the integral from 0 to t_2, instead of from 0 to t_3 is the determining interval for its critical lead mass. Note equation-6 is a general solution including VBR rate profiles.

The profile set given by equation-1, however, allows faster computation. Accordingly, a composite document with continuous media rate R_c, an impulse media size H_I, and a impulse series media rate h_S per time unit t, for a presentation span of T sec, will require a critical lead mass of size:

$$D_{lead} = H + (R_c - B_{freed}) \cdot (T - t) + h\left\lceil \frac{T}{t} \right\rceil + \max\left[t \cdot (R_c - B_{feed}), 0\right] \qquad ...(7)$$

3.6 Lead Segment Composition

Finally, we show the schedule of data segments and allocation of individual streams within D_{lead} and D_{feed}. In multiple parallel streams, excessive preload of one media can potentially result in sub-critical pre-load for the other.

Given the NZC points of the combined rate profile function, we segment the individual media entities into byte segments $B^i(t_j:t_{j+1})$, where i is the media index, and $t_j:t_{j+1}$ is the NGC intervals. We then define a *composite group* as $G_n(j)=\{B^i(t_j:t_{j+1})|$ for all i$\}$. A group contains bytes for all entities that belongs to the same jth NGC segment of the composite document node N_n. the following two rules then applies:

Rule 1: *To ensure critical pre-load all bytes in $G_n(j)$ should be loaded before the bytes in $G_n(j+1)$.*

Rule 2: *However, for lead prefetch, there is no requirement of ordering the bytes within a composite group $G_n(j)$, or between the groups from two nodes $G_n(j)$, and $G_m(j)$, when $n \neq m$.*

Consequently, within the same segment interval, the streams are ordered according to domain specific considerations such documents like stock ticks, can be left for fresh load, while any stored video can be loaded at lead segment.

4. System Model

4.1 System Description

Below we briefly describe the Proxy and Server mechanics for link statistics estimation, partial document fetch and bandwidth partitioning.

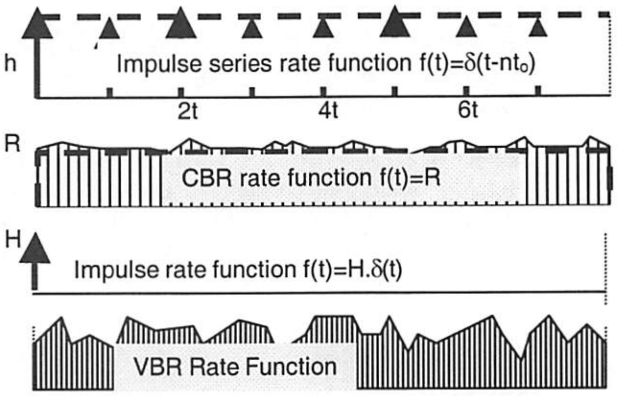
Link Statistics Collection: Statistics about links can be collected via client proxy that embeds link source in the HTTP 1.1 Request-Header Referrer field [11] each time when it issues a HTTP GET request. A server plug-in can track it and resolve the intra-server link references from the referrer id. A group of coop servers then can further resolve the remaining dangling links in batch mode by threshold driven periodic data exchange. The list of embedded link, profile parameters and access statistics for hot documents are can then be stored in a database called *hot-prefetch database* by the origin's stat server.

Statistics Propagation: There are quite a few strategy possible. A simple choice is to use a separate **GET_statistics** request method using HTTP reserve pool. Given an URL in this method the server plug-in can respond with the statistics stored in the *Hot-prefetch database.*

Partial prefetch: The conditional range GET mechanism of HTTP 1.1 (If-Range header, Range and Content-range, Response Code 206 Partial Content) can be used for specifying requests for media segments.

Bandwidth Partition Approximation: Bandwidth partitioning is not available in the current QoS-less Internet. Thus, instead a technique of byte proportioning per time segment basis can be used--

effectively creating the same result. Since all presentations are NZC segmented, first the Range GET requests have to be queued for each element from the current stream node based on its NZC time segments. Within each NZC segment, then the waiting Range GET requests have to be proportionately interleaved for the lead byte-segments earmarked for pre-load. The exact delivery time of the prefetch segments - as long as it is within the correct NZC segment, should not matter. Since, these do not require immediate presentation. However the feed requests are to be placed at the top of queue within each NGC segment. Thus, in effect stream data will enjoy a little prefetch.



h
Impulse series rate function f(t)=δ(t-nt₀)
2t
4t
6t
R
CBR rate function f(t)=R
H
Impulse rate function f(t)=H.δ(t)
VBR Rate Function

Fig-5 shows the profile for four typical traffic types given in equation set (5).

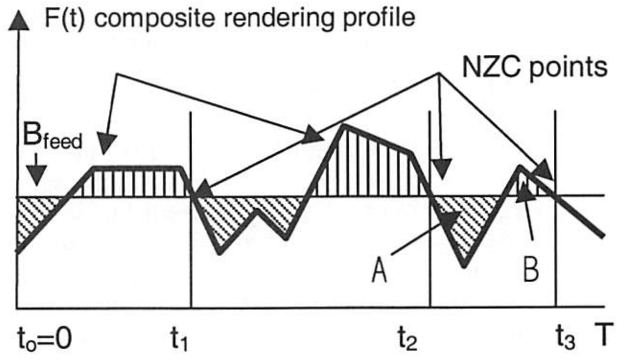


Fig-6 shows how the composite rendering profile can be rate for determining *critical lead mass*. There are three NZC points. In this example, the determining point is the middle one since mass A is larger than mass B.

4.2 Prefetch Mechanism

Now, we briefly describe the design of the prefetch scheme based on the above results. First, in the server side, the prefetch mechanism gathers the composite node statistics- the profile parameters and the access statistics compiled from previous references, in the hot prefetch database.

On the proxy side, when it detects a new *user-agent*, it initializes a new prefetch session for tracking its roaming sphere. The prefetch algorithm then computes $G(V_G,E_G)$ by recursive polling of the links and using a small cut-off threshold ε.

The prefetch mechanism then first computes the critical lead mass for each node on it. Based on this estimate it then determines the priority of the nodes according to the equation set-3(a)-(c).

It also orders the lead byte segments of the candidate nodes according to their group order using rule-1. Finally, the segments within the group are ordered according to rule-2 domain considerations. In our simulation, we rank the continuous media segments with higher priority and those of impulse series with the lowest to ensure late transmission of dynamic media. The selected and ordered prefetch segments are then queued in the fetch queue. The segments that are not in the lead mass however, may also be ordered according to the rules 1 and 2 but are placed on a separate dormant queue.

The current fetch queue is then merged with the feed queue that contains the segment schedule of the streaming segment of the current anchor node. The loading mechanism then generates the Range GET request accordingly.

The loading order remains valid until the current anchor node is read. Upon completion of reading node N, a new node is traversed, and the fetch and feed queues are reorganized. The subsequent evaluation is incremental. A new anchor point changes only the conditional probabilities.

The optimum ranking analysis presented here does not make any assumption about the prediction methods for estimating the link transition probability distributions and it should be applicable with any of the proposed methods [14,20,21]. These methods have varying degree of stability and prediction errors, and computational cost. However, for our simulation our design choice was the simplest one - the access frequency analysis based transition probability estimation. However, as can be seen later, we emphasized on testing the efficacy of the ranking procedure for varying prediction errors.

5. Simulation Results

To characterize the performance of the proposed scheme, we used statistically generated data set rather than server trace. We needed the composition and profile. Also we wanted to stress test the method under varying controlled conditions. Although, trace driven data provides detailed information about a

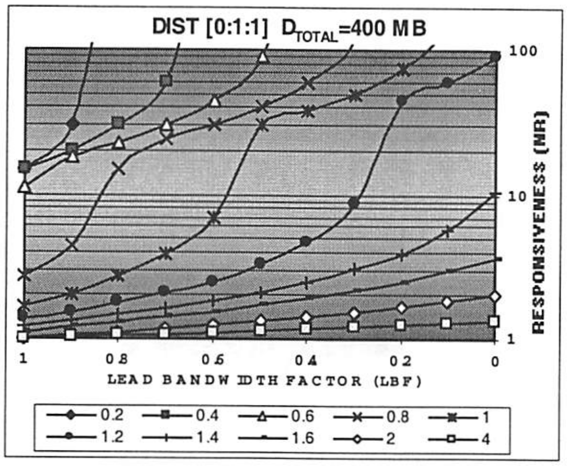


Fig 7(a)

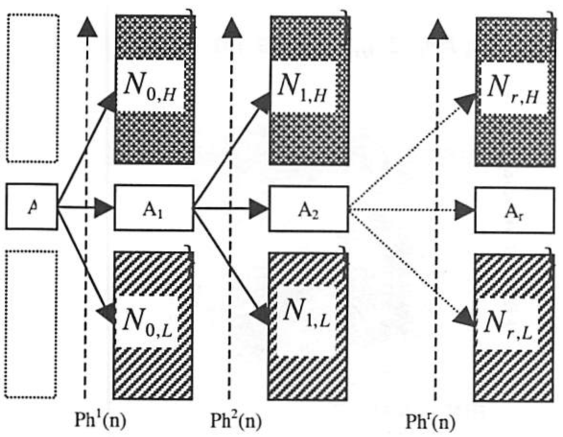


Fig-8 The ordered set of nodes at each state.

particular run, but the controllability of parameters is inadequate. Also the mix and distribution of web-page composition also seems to be changing significantly in short years. Consequently, for this particular work we found it more appropriate to generate broad range of hyperspace data with distinct and varying statistical properties and observe how the proposed method will perform in each of these conditions.

The objective of our first experiment is to observe how the introduction of partial prefetching -- particularly the relative bandwidth allocation between the fetch and feed channel effects. We were also curious to see how the relative rendering speed of various media types (such as text reading speed, play rate for audio or video) will impact the performance.

Consequently, in this first experiment we generated a random set of nodes (composite documents) each with a parent HTML impulse document (containing links to others) and a set of embedded CBR media. We limited the maximum links per node to 10. The underlying algorithm further pruned links with below ε low transition frequency and effectively considered only about 1-4 links for prefetch. HTML documents are given fixed sizes (H^i_S) . We generated the sizes for the CBR media and link transition probability using normal distribution. For a given link bandwidth we then varied the *rendering rate* (h^i_l) for the media as a control variable. Also, since our focus was to track the performance improvement only due to prefetching, we disbarred caching in the simulation. Thus, with each move to a new anchor, all nodes became prefetchable again. Fig-7 plots the observed reduction of the response delay for this experiment. It plots the *responsiveness* (lag-time with active prefetching normalized by that without prefetching). It shows the factor (y-axis) by which the lag time improves with respect to the percentage of bandwidth (x-axis) assigned to the fetch channel-- called *lead bandwidth factor* (LBF) We also plotted the ratio of *network bandwidth* to the *rendering rate*-- called *normalized rendering rate* (nRR). The curves show the performance when the *normalized rendering rate* varies from .2 (slow reading media) to 8 (fast playing media). As can be noted first, that with the inclusion of just prefetching (with 100% bandwidth assigned to the fetch channel or LBF=1.0), the read-time lag can be potentially decreased by a factor of 1-2 (shown by the left-most points) for media rendering rates 4-.8. However, as more bandwidth is set aside for streaming the improvement factor jumps. For example, it goes all the way up-to to 2-15+ times when 50% bandwidth issued for streaming. Not only that for media with slower rendering characteristics (such as text), the improvement is even sharper.

As indicated, the success of any prediction dependent algorithm depends on prediction accuracy. The next experiment we performed was to test how the proposed scheme fares against various levels of prediction discrepancies.

To model the problem, we let the user walk through a chain of anchor points, not necessarily always following the most probable transition path (the link with highest estimated transition probability or priority). However, we made the system to strictly following the ranking given by the equations 3(a)-(c) and 6(a)-(b). At the end of the run, for plotting, at each anchor point A_i, we grouped the nodes of the roaming-spheres into three groups. One with just one node-- the node which was actually traversed $\{a_{i+1}\}$(becoming the next anchor), the other with the nodes those received higher priority than the anchor-- node set $N_{i,H}$, and the third set with the nodes with lower priority than the anchor node--- set $N_{i,L}$. Fig-8 explains the grouping in for several successive anchor phases. We then aggregated the cumulative lead size of each set and observed the delay for various distributions of the bytes in these three sets. We denote the sizes respectively by $|N_{i,H}|$=H, $|N_{i,H}|$=L and $|a_i|$=A.

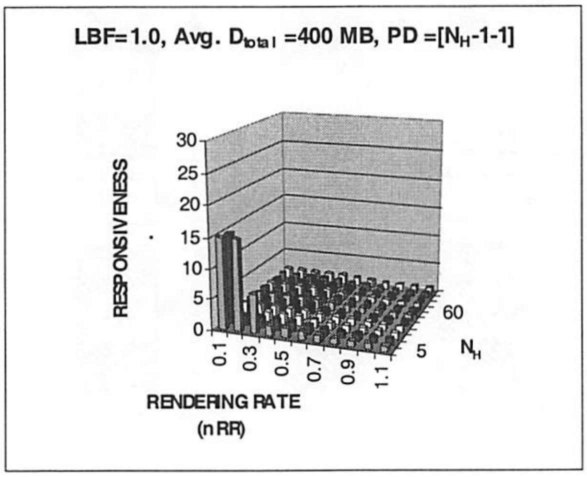


Fig 9(a)

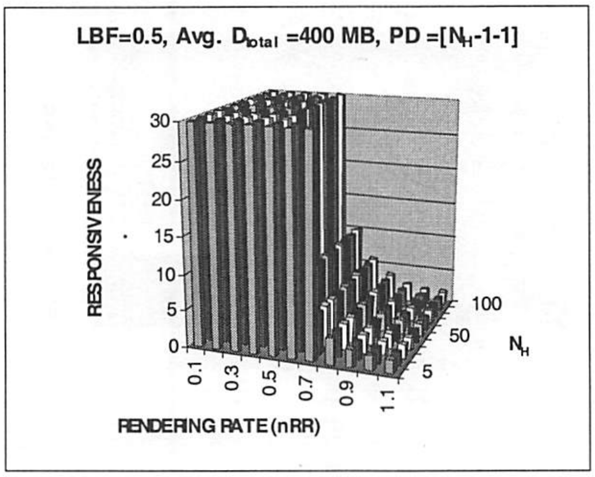


Fig 9(b)

(However, these are not necessarily the actual bytes prefetched, see next experiment for this). We modeled the error by a *prediction distribution ratio* H:A:L denoting the relative sizes of these three sets. We then conducted the previous experiment for various values of H, A and L. Fig-8 shows this grouping for each anchor stage.

First we present the performance for prefetch only model with entire bandwidth allocated to fetch (LBF=1.0). The 3D plot of Fig-9(a) shows the observed *responsiveness* (z-axis) both against the variation of *normalized rendering rate* (x-axis) and the *prediction distribution ratio* (y-axis), where the bytes prefetched before anchor (H) reaches all the way 100 times than the anchor bytes from zero, for the scheme with only prefetch (H had more impact than L). We observed, the system remains effective with high responsiveness (by a factor of +10) only when the rendering is slow (nRR less than 0.3) and the prediction error is small (H≈2).

We then repeated the experiment with partial-prefetch enabled. Fig-9(b) plots the same experiment with LBF=0.5. As can be seen the performance increases dramatically now. Not only the responsiveness improves (factor more than 15), it remained quite high for even relatively faster rendering resources (nRR around 0.7). It also remained immune to quite high level of prediction error (H>>2).

This dramatic result is not unexpected. The reason can be tracked by observing the background line loads. How much of the scheduled bytes (=H+A+L) are actually fetched depends on when the user moves to next anchor (Δ_{i+1}). If at this moment $N_{i,H}$, were loading then it would skip the remain and begin loading Δ_{i+1}. On the other hand, if it were in $N_{i,L}$, it would ignore its remaining part and render Δ_{i+1}. We define *load factor* as the ratio of the actual bytes prefetched to the bytes actually read (size anchor sequence). Fig-10(a) and (b) respectively trace this *load factor* (z-axis) for these same two cases (with and without partial prefetch). As can be noted that streaming enabled prefetching dramatically reduced the background line load--almost near one in large part of the graph.

The above results show how a streaming incorporated

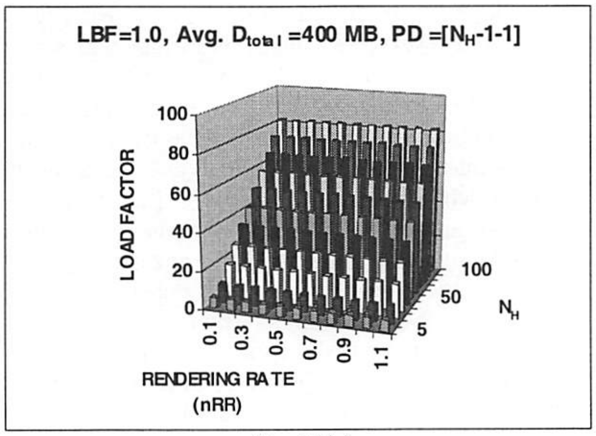


Fig 10(a)

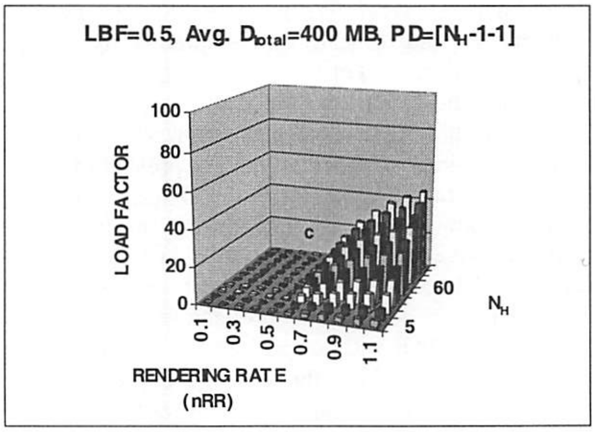


Fig 10(b)

active prefetching can significantly improve the responsiveness of a cache system. To summarize, the performance without partial prefetch, matched those reported by other researchers. However, the partial prefetch resulted in dramatic improvement. Careful reduction of data-bytes per node, in effect allowed information from more candidate nodes to be pre-fetched, given the same bandwidth and local storage.

6. Conclusions & Current Work

The prefetch prediction model reduces access lag for new references. It is quite appealing for the Web, and has attracted significant research interest. However, just prefetching seems to be limiting because of excessive transfer of unused bytes. This has been reported in few of the other research as well.

In this paper, we have suggested an innovative method that integrates a concept similar to streaming, and greatly reduces this waste, by prefetching only a carefully calculated amount. In this paper, we have outlined the method for a web composed of **composite documents**, which are becoming more ubiquitous in performance sensitive sites. We also presented the analytical considerations backing the design. Also, in this paper, we have presented simulation results based on statistical models that projects the scheme's performance under varying conditions. We think, such data optimization in prefetch is very critical because poorly done prefetch can adversely impact overall network performance.

The paper's principle focus is on the **data scheduling**. However, there are many other issues, which need further investigation. As indicated, the estimation of link transition probability, whether it is from conventional access log [21], or from explicit message exchange [8] as shown here, needs much more analysis.

In the systems design section we have only outlined implementation techniques for steps which appears non-trivial at first look- including link transition statistics collection or bandwidth partitioning on QoS less Internet. However, the actual deployment of any prefetch system, including ours will require more work on protocols. It seems that any prefetch will require prediction, which in turn will require exchange of document statistics between servers and server and clients. Also, the tracking ability of nearby hyper-graph, as demonstrated here will be useful. Also, interesting is protocol enhancements for partial document transfer for HTML, XML, RTSP entities [6]. Also, we did not present any prefetch algorithm, rather focused just on the ranking.

Within the scope of this paper, we deliberately switched off any caching. However, we are also currently studying the impact when caching is added, the results of which will be presented in a forthcoming submission. The points of interest are how the cache parameters --- cache size, media classification, and discard policies, interplays here.

As a part of our ongoing research, we are currently investigating an active net deployable prefetch proxy module, which can be dynamically launched as an active proxy inside network. The work is currently being funded by DARPA Research Grant F30602-99-1-0515 under its Active Network initiative.

7. References:

[1] Marc Abrams, Charles R. Standridge, G. Abdulla, A. Edward, Fox and S. Williams, Removal policies in network caches for World-Wide Web documents, ACM SIGCOMM , Stanford, CA, 1996, pp 293-305.

[2] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. Proc. USENIX Technical Conf., CA, January 1997, pp. 289-303

[3] High-Performance Web Caching White Paper, 1998 CacheFlow Inc. [Retrieved on June 8th, 2000 from URL http://www.cacheflow.com/technology/whitepapers/web.cfm]

[4] E. Cohen and H. Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. Procs. of the IEEE INFOCOM 2000, Tel-Aviv, Israel, March 2000.

[5] M. Crovella, P. Barford, The Network Effects of prefetching, Proc. Of IEEE INFOCOM 1998, San Francisco, USA, 1998.

[6] A. Dan and D. Sitaram, Multimedia caching strategies for heterogeneous application and server environments, Multimedia Tools and Applications, vol. 4, pp.279-312, May 1997.

[7] F. Douglis, A. Feldman, B. Krisnamurty and J. Mogul, Rate of Change and Other Matrices: A Live Study of the World Wide Web, Proc. Of USENIX Symposium on Internet Technology and Systems, Barkeley, December 1997, pp-147-158.

[8] D. Duchamp. Prefetching Hyperlinks. Proceedings of the USENIX Symposium on

Internet Technologies and Systems, Colorado, USA, October 1999. [Http://www.usenix.org/events/usits99].

[9] Li Fan, Pei Cao, and Quinn Jacobson. Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance. Procs. of the ACM SIGMETRICS'99, Atlanta, Georgia, May 1999.

[10] A. Feldmann, R. Caceres, F. Douglis, G. Glass and M. Rabinovich, Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments, Proceedings of INFOCOM 99, 1999.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk & T. Berners-Lee, Hypertext Transfer Protocol HTTP/1.1, RFC 2068, January 1997.

[12] S. Gruber, J. Rexford, and A. Basso, Design considerations for an RTSP-based prefix caching proxy service for multimedia streams, Tech. Rep. 990907-01, AT&T Labs - Research, September 1999.

[13] Grosso, Paul, Daniel Veillard, XML Fragment Interchange, W3C Working Draft 1999 June 30, [Retrieved from: http://www.w3.org/1999/06/WD-xml-fragment-19990630.html]

[14] Q. Jacobson, Pei Cao, Potential and Limits of Web Prefetching Between Low-Bandwidth Clients and Proxies, 3rd International WWW Caching Workshop, Manchester, England, June 15-17 1998.

[15] J. Jung, D. Lee, and K. Chon, Proactive Web Caching with Cumulative Prefetching for Large Multimedia Data. Procs. of the 9th International World Wide Web Conference, Amsterdam, Netherlands, May 2000.

[16] Javed I. Khan, Ordering Prefetch in Trees, Sequences and Graphs, Technical Report 1999-12-03, Kent State University, [available at URL http://medianet.kent. edu/ technicalreports.html, also mirrored at http:// bristi.facnet.mcs.kent.edu/~javed/medianet]

[17] Javed I. Khan, Active Streaming in Transport Delay Minimization, Workshop on Scalable Web Services, Int. Conf. on Parallel Processing, Toronto, August 2000, pp95-102.

[18] T. Kroeger, D. D. E. Long & J. Mogul, Exploring the Bounds of Web Latency Reduction from Caching and Prefetching, Proc. Of USENIX Symposium on Internet Technology and Systems, Monterey, December 1997, pp-319-328.

[19] NLANR, Proxy cache log traces, December 1999, ftp://ircache.nlanr.net/Traces/.

[20] T. Palpanas and A. Mendelzon,, Web Prefetching Using Partial Match Prediction, WWW Caching Workshop, San Diego, CA, March 1999

[21] P. Pirolli and J. E. Pitkow, Distributions of surfers' paths through the World Wide Web: Empirical characterizations, Jounral of World Wide Web, 1999, v.1-2, pp29-45

[22] J. Rexford, S. Sen, and A. Basso, A smoothing proxy service for variable-bit-rate streaming video, in Proc. Global Internet Symposium, December 1999.

[23] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, RTP: A transport protocol for real-time applications,' RFC 1889, January 1996.

[24] H. Schulzrinne, A. Rao, & R. Lanphier, Real Time Streaming Protocol (RTSP), RFC: 2326 , April 1998 [Retrieved on June 8th, 2000 from URL ftp://ftp.isi.edu/in-notes/rfc2326.txt]

[25] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In Proceedings of the IEEE INFOCOM'99 Conference, 1999.

[26] Squid Web Proxy Cache ,http://www.squid-cache.org, 1999.

[27] Z. Wang and J. Crowcroft, Prefetching in the World Wide Web. in procs. of IEEE Global Internet, London, UK, 1996.

Modular and Efficient Resource Management in the *Exedra* Media Server

Stergios V. Anastasiadis* Kenneth C. Sevcik* Michael Stumm†

*Department of Computer Science
†Department of Electrical and Computer Engineering
University of Toronto
{stergios@cs,kcs@cs,stumm@eecg}.toronto.edu

Abstract

Explosive growth in online services has recently renewed the interest for building modular and efficient network server systems. System design complications coupled with excessive expectations from technological progress previously discouraged the development of media servers efficiently supporting video streams with variable bit rates. In this paper, we describe the design of a distributed media server architecture, and the implementation details of a prototype. Native support is provided for variable bit rate streams, by considering their special features in the resource management policies. We identify several problems, and propose new approaches for scheduling the playback requests, organizing the memory buffers, allocating the storage space, and structuring the disk metadata. We justify several of our decisions with comparative performance measurements using both synthetic benchmarks and actual experiments with variable bit rate MPEG-2 streams over SCSI disks.

1 Introduction

Variable Bit Rate (VBR) video streams are estimated to be smaller (by 40% or more) than *Constant Bit Rate (CBR)* streams of comparable quality [10, 13]. Correspondingly, media servers supporting VBR streams can be expected to concurrently serve more users than their CBR counterparts, due to reduced requirements for disk space, disk bandwidth, buffer space, and network bandwidth.

Nevertheless, most of the existing experimental or commercial media servers we are aware of can only support CBR streams [3, 12]. Alternatively they store VBR streams using either peak rate resource reservations that may reduce resource utilization but not increase server capacity, or statistical QoS guarantees that allow the system to occasionally be overloaded and discard data [14, 18, 24]. The approach of retrieving VBR streams using constant rates, popular with lower bit rate streaming applications, might not solve the general problem either due to arbitrarily large playback initiation latency or client buffer space that it can require with higher quality streams [20, 23].

We believe that the above issues remain unresolved, because the feasibility and potential advantages of deterministically supporting VBR streams over multiple disks have not been demonstrated yet. In this paper we describe *Exedra*, a video server we designed and built, that uniquely combines the following key features:

1. native support for VBR streams with deterministic QoS guarantees,
2. striping of stream data across multiple disks,
3. detailed reservation of system resources over time.

In describing our prototype, we focus on important design tradeoffs with respect to dispatching the playback requests, organizing the memory buffers, allocating the storage space, and structuring the disk metadata. We justify several of our design decisions with comparative performance evaluation using our implementation with synthetic benchmarks and actual variable bit rate MPEG-2 streams over SCSI disks.

The remainder of the paper is structured as follows. In Sections 2 and 3, we go over the components of the proposed media server architecture and our prototype, as well as related design alternatives. In Section 4 we introduce the experimentation environment that we used. In Section 5, we present the results of our performance experiments. In Section 6, we compare our system with previous related work, and in Section 7 we summarize our conclusions.

2 Server Architecture

In this section we give a high-level overview of an architecture, and describe the underlying assumptions of the system operation. In addition, we introduce the components of the server and their functionality, a method for efficiently allocating the disk space, and the model for reserving resources at the server side.

2.1 Overview

Exedra[1] is a distributed media server system based on standard off-the-self components for data storage and transfer. Video streams are stored on multiple disks, compressed according to the MPEG-2 specification, with constant quality quantization parameters and variable bit rates. Multiple clients with appropriate stream decoding capability send playback requests to the server and receive stream data via a high-speed network, as shown in Figure 1.

We assume that the system operates using the server-push model. When a playback session starts, the server periodically sends data to the client until either the end of the stream is reached, or the client explicitly requests suspension of the playback. The server-push model facilitates quality of service enforcement at the server side, when compared to a client-pull model [25]. We also assume that data transfers occur in rounds of fixed duration T_{round}. In each round, an appropriate amount of data is retrieved from the disks into a set of server buffers reserved for each active client. Concurrently, data are sent from the server buffers to the client through the network interfaces. Round-based operation is typically used in media servers in order to keep the reservation of the resources and the scheduling-related bookkeeping of the data transfers manageable.

The large amount of network bandwidth needed for this kind of service requires that the server consists of multiple components, connected to the high-speed network through different network interfaces [3]. The amount of stream data periodically sent to the client is determined by the decoding frame rate of the stream and the resource management policy of the network. A reasonable policy would send to the client during each round the amount of data that will be needed for the decoding process of the next round; any other policy that does not violate the timing requirements and buffering constraints of the decoding client would also be acceptable.

[1]Exedra: architectural term meaning semicircular or rectangular niche (orig. Greek).

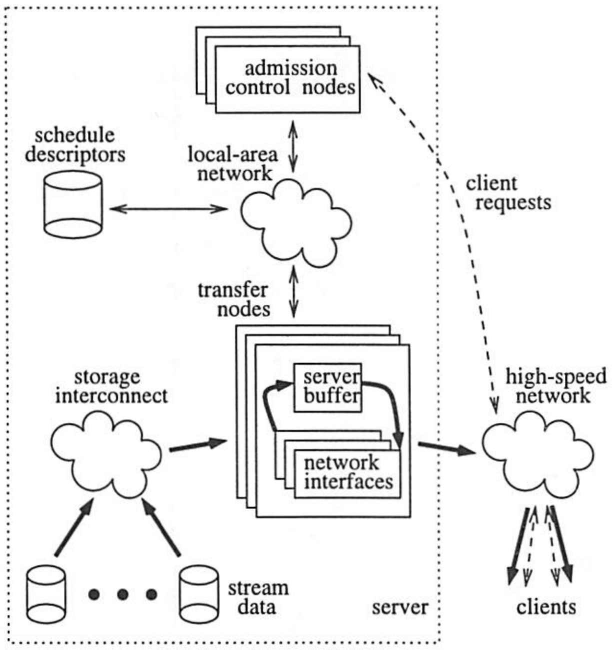


Figure 1: In the *Exedra* media server, stream data are retrieved from the disks and sent to the clients through the Transfer Nodes. Both the admission control and the data transfers make use of stream scheduling information maintained as a set of schedule descriptors.

2.2 System Components

The stream data are stored across multiple disks. As shown in Figure 1, each disk is connected to a particular *Transfer Node* through the *Storage Interconnect*, which could be either i) standard I/O channel (e.g. Small Computer System Interface), ii) network storage equipment (e.g. Fibre-Channel [6]), or iii) a general purpose network (as with Network-Attached Secure Disks [9]). Recent research has demonstrated that it is possible to offload file server functionality to network-attached disks [9]. Although we believe that our design could be extended in a similar way, we leave the study of this issue for future work.

The Transfer Nodes are standard, off-the-shelf computers responsible for scheduling and initiating all data transfers from the attached disks to the clients. Data arriving from the disks are temporarily staged in the *Server Buffer* memory of the Transfer Node before being sent to the clients. We assume that the system bus bandwidth (such as the Peripheral Component Interconnect) is a critical resource within each Transfer Node that essentially restricts the number and the capacity of the attached network and I/O channel interfaces.

Playback requests arriving from the clients are initially directed to an *Admission Control Node*, where it is decided whether sufficient resources exist to activate the requested playback session either immediately or within a few rounds. The computational complexity of the general stream scheduling problem is combinatorial in the number of streams considered for activation and the number of reserved resources [8]. However, we make the practical assumption that the users can only wait a limited number of rounds before actual playback starts. This limits the number of future rounds considered for playback initiation, and permits us to use a simpler scheduling algorithm with complexity linear in the number of rounds of each stream and the number of reserved resources.

For example, we can assume rounds of one second, a two hour stream, and a single node system with two disks and one network interface. Then, the maximum number of numerical comparisons required is equal to the stream length (7200) multiplied by the number of resources (4). This computation requirement corresponds to a worst case general case. It can be significantly relaxed for the particular data striping method we use in practice, since not all the resources are involved in each round. If the test fails, it has to be repeated for each extra future round considered for playback inititiation. Depending on the expected load and the required detail of resource reservation, the admission control process might still become a bottleneck. In that case, the admission control could be distributed across multiple processors as shown in Figure 1, taking into account non-trivial concurrency control issues that arise. If a new playback request is accepted, commands are sent to the Transfer Nodes to begin the appropriate data accesses and transfers.

The amount of stream data that needs to be retrieved during each round from each disk is stored in a *Schedule Descriptor*. The descriptor also specifies the buffer space required and the amount of data sent to the client by the Transfer Nodes during each round. It is possible that two or more schedule descriptors are available for the same stream with distinct requirements. The scheduling information is generated when a stream is first stored and is used for both admission control and for controlling data transfers during playback. Since this information changes infrequently, it can be replicated to avoid potential bottlenecks.

2.3 Stride-Based Disk Space Allocation

In our system, we use a new form of disk space allocation, called *stride-based allocation* [1], in which

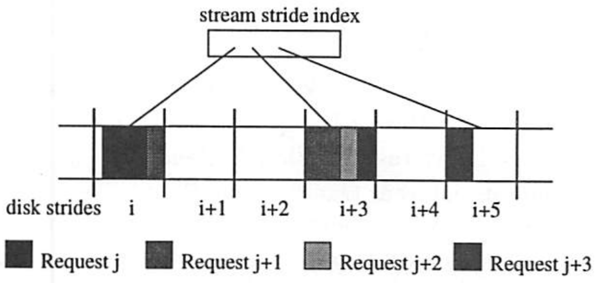


Figure 2: The stride-based allocation of disk space is shown on one disk. A stream is stored in a sequence of generally non-consecutive fixed-size strides with a stride possibly containing data of more than one round. Sequential requests of one round are smaller than the stride size and thus require at most two partial stride accesses.

disk space is allocated in large, fixed-sized chunks (*strides*) that are sequentially allocated on the disk surfaces. Strides are chosen larger than the maximum stream request size per disk during a round. (This size is known in advance for stored streams.)

Stride-based allocation has a number of advantages over schemes that are used in other systems [5, 16, 24]. It sets an upper-bound on the estimated disk access overhead, since at most two partial stride accesses will be required to serve the request of a stream on each disk in a round. It eliminates external fragmentation, while keeping internal fragmentation negligible because of the large size of the streams, and because a stride may contain data of more than one round (see Figure 2). When a stream is retrieved, only the requested amount of data is fetched to memory and not the entire stride.

2.4 Reservation of Server Resources

A mathematical abstraction of the resource requirements is necessary for scheduling purposes. Consider a system consisting of N network interfaces, D disks, and Q transfer nodes.

The stream *Network Striping Sequence*, $\mathbf{S_n}$, of length L_n defines the amount of data, $S_n(i,u)$, $1 \leq i \leq L_n$, $0 \leq u \leq N-1$, that the server sends to a particular client through network interface u during round i. Similarly, the *Buffer Striping Sequence* $\mathbf{S_b}$ of length $L_b = L_n + 1$ defines the server buffer space required on node q, $S_b(i,q)$, $0 \leq i \leq L_b$, $0 \leq q \leq Q-1$ during round i. Each stride is a sequence of logical blocks with fixed size B_l, which is multiple of the physical sector size B_p of the disk. Both disk transfer requests and memory buffer reservations are specified in multiples of the block size B_l. The *Disk Striping Sequence* $\mathbf{S_d}$ of length $L_d = L_n$ determines the amount of data, $S_d(i,k)$, that are retrieved from disk k, $0 \leq k \leq D-1$, in round i, $0 \leq i \leq L_d - 1$.

We assume that disk k, $0 \leq k \leq D-1$, has edge to edge seek time $T^k_{fullSeek}$, single track seek time $T^k_{trackSeek}$, average rotational latency T^k_{avgRot}, and minimum internal transfer rate R^k_{disk}. The stride-based disk space allocation policy enforces an upper bound of at most two disk arm movements per disk for each client per round. The total seek distance can also be limited using a CSCAN disk scheduling policy. We assume that seek latency is always a linear function of the seek distance.[2] Head settling time is accounted for through the single-track seek time parameter of the disk specification. We quantify later the accuracy of these approximations.

Let M_i be the number of active streams during round i of the system operation. Also the playback of stream j, $1 \leq j \leq M_i$, is initiated at round l_j of system operation. Then, the total access time on disk k in round i of the system operation will have an upper-bound of:

$$\begin{aligned} T_{disk}(i,k) =& 2T^k_{fullSeek} + 2M_i \cdot (T^k_{trackSeek} + T^k_{avgRot}) \\ & + \sum_{j=1}^{M_i} S^j_d(i - l_j, k)/R^k_{disk} \qquad (1) \end{aligned}$$

where $\mathbf{S^j_d}$ is the disk striping sequence of client j. $T^k_{fullSeek}$ is counted twice due to the disk arm movement from the CSCAN policy, while the factor two in the second term is due to the stride-based allocation. The reservations of transfer time on each network interface and buffer space on each transfer node are more straightforward, and are based on the Network Striping Sequence and Buffer Striping Sequence, respectively.

2.5 Variable-Grain Striping

With *Variable-Grain Striping*, stream files are stored on disks such that the data retrieved during a round for a client are always accessed from a single disk round-robin. Comparison with alternative striping techniques has shown significant performance benefits when using Variable-Grain Striping [1, 5, 21], and this is the method that we use in the present study.

3 Prototype Implementation

We have designed and built a single-node multiple-disk media server prototype in order to evaluate the

[2]For short seeks, seek latency is known to depend on the square root of the seek distance though [22].

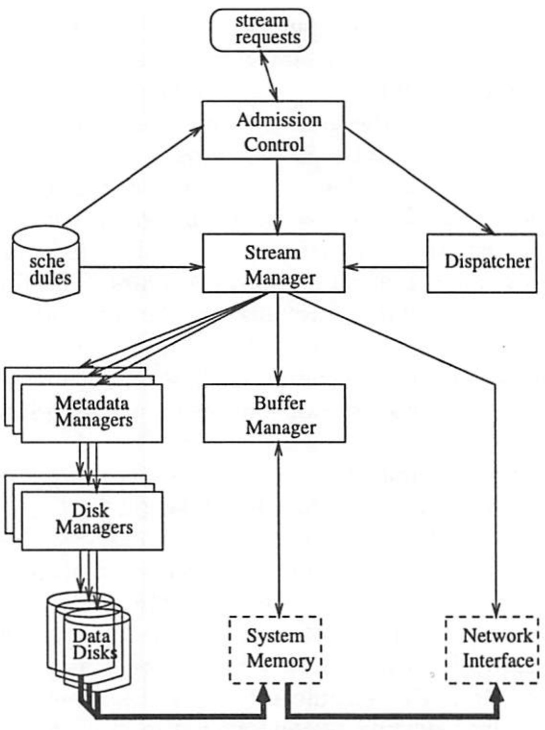


Figure 3: System modules in the *Exedra* prototype implementation.

resource requirements of alternative stream scheduling techniques. The modules are implemented in about 12,000 lines of C++/ Pthreads code on AIX4.2. The code can be linked to the University of Michigan DiskSim disk simulation package [7], which incorporates advanced features of modern disks, such as on-disk cache and zones, for obtaining disk access time measurements. The code can also directly use hardware disks through their raw interface for full data transfers. The stream indexing metadata are stored in the Unix file system as regular files, and during operation are kept in main memory.

The basic responsibilities of the media server include file naming, resource reservation, admission control, logical to physical metadata mapping, buffer management, and disk and network transfer scheduling (Figure 3). With appropriate configuration parameters, the system can operate in several modes to allow different levels of detail in our evaluation analysis. In *Admission Control* mode, the system receives playback requests, does admission control and resource reservation, but no actual data transfers take place. In *Simulated Disk* mode, all the modules become functional, and disk request processing takes place using the specified DiskSim [7] disk array.[3] In *Full Operation* mode, the system ac-

[3]This mode is not used in the current study.

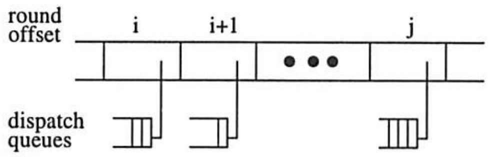


Figure 4: A circular vector of dispatch queues keeps track of admitted streams yet to be activated. The dispatch queue consists of notification records for activating the streams in the corresponding rounds.

cesses hardware disks and transfers data to clients.

We now describe in more detail the modules from our implementation that are responsible for admission control, metadata management, disk scheduling, and buffer management.

3.1 Admission Control and Dispatching

The admission control module uses circular vectors of sufficient length to represent the allocated disk time, network time, and buffer space, respectively. On system startup, all elements of disk time vectors are initialized to $2 \cdot T_{fullSeek}$, while the network time and buffer space vector elements are set to zero. When a new stream request arrives, the admission control is performed by checking the requirements of the stream against currently available resources. In particular, the total service time of each disk in any round may not exceed the round duration, the total network service time on any network interface may not exceed the round duration, and the total occupied buffer space on any transfer node may be no larger than the corresponding server buffer capacity.

If the admission control test is passed, then the resource sequences of the stream are added to the corresponding system vectors managed by the module, and the stream is scheduled for playback. In addition, notification records for the accepted request are inserted into the corresponding dispatch queues (generally residing on each transfer node) at the appropriate offset from the current round. When an upcoming round becomes current, the notification records are used for activating the stream and starting its data transfers (Figure 4).

3.2 Metadata Management

Stream metadata management is organized in a layer above disk scheduling. It is responsible for disk space allocation during stream recording, and for translating stream file offsets to physical block locations during playback. The stream metadata are maintained as regular files in the host OS (of each transfer node, in the general case), while stream data are stored separately on dedicated disks. The storage space of the data disks is organized in strides, with a bitmap that has a separate bit for each stride. A single-level directory is used for mapping the identifier of each recorded stream into a direct index of the corresponding allocated strides. A separate directory of this form exists for each disk.

When a stream is striped across multiple disks, a stream file is created on each data disk. Each transfer request received by the metadata manager specifies the starting offset in the corresponding stream file and the number of logical blocks to be accessed. With the help of the stream index, each such request is translated to a sequence of contiguous disk transfers, each specifying the starting physical block location and the number of blocks. From the stride-based disk space allocation, it follows that each logical request will be translated into at most two physical contiguous disk transfers.

The decision to create a separate metadata manager for each disk was motivated by our intention to experiment with general disk array organizations, including those consisting of heterogeneous disks. Although the handling of heterogeneous devices may not be necessary in limited size traditional storage systems, it might prove crucial for the incremental growth and survivability of large scalable media storage installations. In our prototype implementation, this feature is fully implemented in a relatively straightforward way as described above.

In order to keep system performance predictable and unbiased from particular disk geometry features, we exercise some control on the disk space allocation pattern. In particular, disk zoning could possibly lead to excessively optimistic or pessimistic data access delays, if we mostly allocated the outer or inner cylinders of the disks. Similarly, contiguous allocation could lead to lower than expected delays in some special cases (such as when streams are stored on a single disk with a very large on-disk cache). However, low-level disk geometry is generally not disclosed by the disk manufacturers, and the above features are not explicitly considered by the system in any sophisticated way. Therefore, when we allocate strides for a stream within each disk, we try to distribute them across all the zones of the disk.

3.3 Disk Scheduling

The disk management layer is responsible for passing data transfer requests to the disks, after the necessary translation from logical stream offsets to physical block locations in the above layers.

In the *dual-queue CSCAN* disk scheduling that we use, the operation of each disk is managed by a separate pair of priority queues, called *Request Queue* and *Service Queue*, respectively. The two queues, although structurally equivalent, play different roles during each round. At the beginning of each round, data transfer requests for the current round are added asynchronously into the request queue of each disk, where they are kept sorted in increasing order of their starting sector location.

When all the requests have been gathered (and the corresponding disk transfers of the previous round completed), the request queue of each disk is swapped with the corresponding service queue. Subsequently, requests from the service queue are synchronously submitted to the raw disk interface for the corresponding data transfers to occur. The two-queue scheme prevents new requests from getting service before those of the previous round complete. This keeps the system operation more stable in the rare (yet possible) case that the disk busy time in a round slightly exceeds the round duration.

3.4 Buffer Management

The buffer management module keeps the server memory organized in fixed size blocks of B_l bytes each, where B_l is the logical block size introduced earlier. Buffer space is allocated in groups of consecutive blocks. From experiments with raw interface disk accesses, we found that non-contiguity of the memory buffers could penalize disk bandwidth significantly on some systems. Although this might be attributed to the way that scatter/gather features of the disk controller are used by these systems, we found the allocation contiguity easy to enforce.

For the allocation of buffer blocks we used a bitmap structure with an interface that can support block group requests. Deallocations are allowed on a block by block basis, even though entire block groups are acquired during allocation. This last feature allows more aggressive deallocations.

In our design, we do not cache previously accessed data, as is typically done in traditional file and database systems. Although related research has developed data caching algorithms for constant rate streams, we found that similar support for variable bit rate streams would introduce several complications, especially in the admission control process. Instead, we assume that data transfers are done independently for each different playback [3].

Paging of buffer space is prevented by locking the corresponding pages in main memory. Although several Unix versions (e.g. HP-UX, Irix, Solaris) and Linux make the *mlock* system call available for this purpose, AIX does not. Instead, we exported the *pinu* kernel service through a loadable kernel module and used that.

Seagate Cheetah ST-34501W	
Data Bytes per Drive	4.55 GB
Average Sectors per Track	170
Data Cylinders	6,526
Data Surfaces	8
Zones	7
Buffer Size	0.5 MB
Track to Track Seek(read/write)	0.98/1.24 msec
Maximum Seek(read/write)	18.2/19.2 msec
Average Rotational Latency	2.99 msec
Internal Transfer Rate	
Inner Zone to Outer Zone Burst	122 to 177 Mbit/s
Inner Zone to Outer Zone Sustained	11.3 to 16.8 MB/s
External Transfer Rate	40 MB/s

Table 1: Features of the SCSI disks used in our experiments.

Content Type	Avg Bytes per rnd	Max Bytes per rnd	CoV per rnd
Science Fiction	624935	1201221	0.383
Music Clip	624728	1201221	0.366
Action	624194	1201221	0.245
Talk Show	624729	1201221	0.234
Adventure	624658	1201221	0.201
Documentary	625062	625786	0.028

Table 2: We used six MPEG-2 video streams of 30 minutes duration each. The coefficient of variation shown in the last column changes according to the content type.

4 Experimentation Setup

Our performance measurements were made on an IBM RS/6000 two-way SMP workstation with 233 MHz PowerPC processors running AIX4.2. The system was configured with 256 MB physical memory, and a fast wide SCSI controller to which a single 2GB disk was attached, containing both the system and paging partitions. The stream data are stored on two 4.5GB Seagate Cheetah ST-34501W disks (Table 1) attached to a separate ultra wide SCSI controller.[4] Although storage capacity can reach $73GB$ in the latest models, the performance numbers of the above two disks are typical of today's high-end drives.

We used six different variable bit rate MPEG-2 streams of 30 minutes duration each. Each stream

[4]Note that one megabyte (megabit) is considered equal to 2^{20} bytes (bits), except for the measurement of transmission rates and disk storage capacities where it is assumed to be equal to 10^6 bytes (bits) instead [11].

has 54,000 frames with a resolution of 720x480 and 24 bit color depth, 30 frames per second frequency, and a $IB^2PB^2\ PB^2PB^2PB^2$ 15 frame group of pictures structure. The encoding hardware that we used allows the generated bit rate to take values between 1Mbit/s and 9.6Mbit/s. Statistical characteristics of the clips are given in Table 2, where the coefficients of variation (of bytes per round) lie between 0.028 and 0.383, depending on the content type. We used the MPEG-2 decoder from the MPEG Software Simulation Group for stream frame size identification [17].

Unless otherwise stated, the logical block size B_l was set equal to 16 KB, while the physical sector size B_p was 512 bytes. The stride size B_s in the disk space allocation was set to 2 MB. The total memory buffer size was set to 64 MB, organized in fixed size blocks of 16 KB. In our experiments, data retrieved from the disks are discarded (copied from the buffer to the *null* device at the appropriate round), leaving protocol processing and contention for the network outside the scope of the present study.[5] The round time was set equal to one second.

We assume that playback initiation requests arrive independently of one another, according to a Poisson process. The system load can be controlled by setting the mean arrival rate λ of playback initiation requests. The maximum possible service rate μ, expressed in streams per round for streams of data size S_{tot} bytes, is equal to $\mu = \frac{D \cdot R_{disk} \cdot T_{round}}{S_{tot}}$. Correspondingly, the system load ρ, is equal to $\rho = \frac{\lambda}{\mu} \leq 1$, where $\lambda \leq \lambda_{max} = \mu$. The definition of ρ is used by the experiments that follow, and is justified in more detail elsewhere [1].

When a playback request arrives, the admission control module checks whether available resources exist for every round during playback. The test considers the exact data transfers of the requested playback for every round and also the corresponding available disk transfer time, network transfer time and buffer space in the system. If the request cannot be initiated in the next round, the test is repeated for each round up to $\lceil \frac{1}{\lambda} \rceil$ rounds into the future, until the first round is found, where the requested playback can be started with guaranteed sufficiency of resources. Checking $\lceil \frac{1}{\lambda} \rceil$ rounds into the future achieves most of the potential system capacity as was shown previously [1]. If not accepted, the request is discarded rather than being kept in a queue.

[5]Not including the network protocol overhead in the measurements that follow, allowed us to demonstrate the exact cost of the disk transfers involved, which is our main focus here. On the other hand, experiments that we did using the loopback interface gave results within 5% of those reported.

The experiments are repeated until the half-length of the 95% confidence interval on the performance measure of interest lies within 5% of the estimated mean value. Our basic performance objective is to maximize the average number of active playback sessions that can be concurrently supported by the server.

5 Performance Evaluation

We begin our experiments by examining the potential effects of our buffer organization on the disk throughput. We also investigate implications of the disk space allocation parameters to the disk bandwidth utilization, and compare resource reservation statistics to actual utilization measurements. Finally, we demonstrate that system throughput scales linearly with the amount of resources made available, under the disk striping policy that we use.

5.1 Contiguity of Buffer Allocation

We evaluate the performance of the buffer allocation policy using a synthetic benchmark that we developed for this purpose. We measure the disk throughput for different sizes of I/O requests and degrees of contiguity in the buffer space allocated for each request. Disk requests of a specific size are initiated at different locations uniformly distributed across the disk space. Data are transferred through the raw disk interface to pinned memory organized in blocks of fixed size B_l, similar to our prototype.

In Figure 5(a), we depict the average throughput, when a separate `read()` call is invoked for each buffer block corresponding to a request. We vary both the block size and the size of the request. When the block size is increased from 4 KB to 64 KB, disk throughput changes by a factor of three across the different request sizes. When the request size varies from 64 KB to 4 MB for a particular block size, the throughput increases by more than a factor of two.

In Figure 5(b), we repeat the previous measurements by using the `readv()` system call instead. It takes as parameters the pointer to an array of address-length buffer descriptors along with the size of the array. The array size is typically limited to a small number of buffer descriptors (e.g. IOV_MAX = 16 in AIX and Solaris). For each I/O request, the required number of `readv()` calls is used, with the array entries initialized to the address and length of each buffer block. Although we expected improved performance due to the increased amount of information supplied with each `readv()` call to the OS, the measured throughput was less than half of what

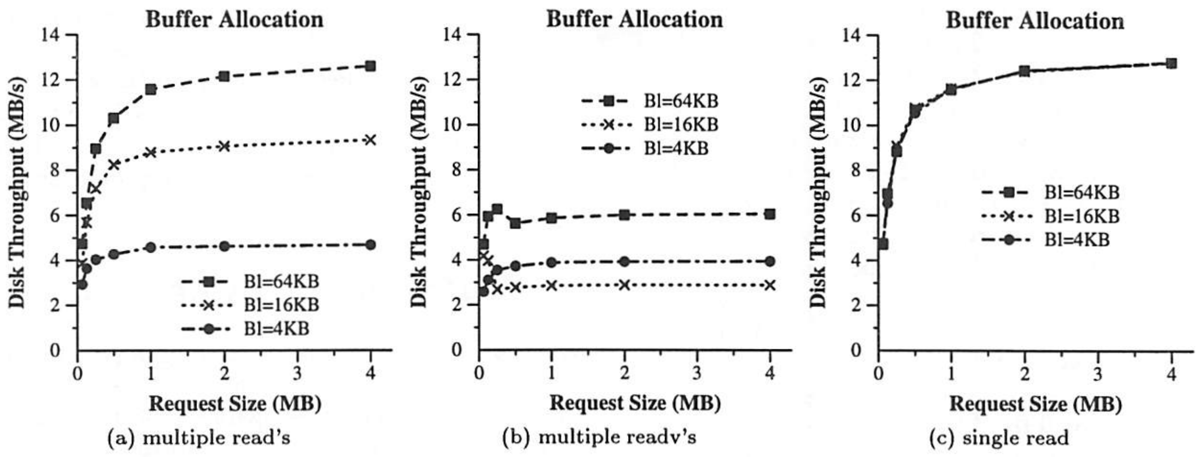


Figure 5: a). When we use a separate disk transfer for each buffer block, disk throughput depends critically on the block size. b) Grouping multiple block transfers into a single call by using `readv()` cuts by more than 50% the achieved disk throughput. c) Invoking a single `read()` for each request keeps disk throughput consistently high, independently of the buffer block size.

we measured with `read()`. Proper explanation for this would probably require internal knowledge of the AIX device drivers that we didn't have. An additional limit is also imposed to the I/O performance due to the small value of the IOV_MAX.

In Figure 5(c), we repeated the previous experiments, by using only a single `read()` call for each request, similar to the way I/O requests are served in our prototype.[6] This policy requires contiguously located buffer blocks in the virtual address space. As expected the sensitivity to the block size B_l disappears. Note that the achieved performance is only slightly higher than that of figure 5(a) with large blocks. However, the block size itself cannot be arbitrarily large; otherwise the benefit from multiplexing requests of different sizes drops, which eventually reduces the number of accepted streams [1]. Since the average size of the disk transfers is about 625,000 bytes in our MPEG-2 clips, from these experiments we can expect the disks to operate at average throughput higher than 11 MB/s, which is consistent with the achievable sustained rate of 11.3 MB/s advertised in the disk specification.

We conclude (for this system) that contiguity in the buffer space allows a relatively small block size to be chosen that guarantees both the support of a large number of streams and efficient disk transfers. This simplifies the performance tuning of the system. One disadvantage is the complexity introduced by having to manage buffer block ranges instead of fixed buffers. In addition, buffer space fragmentation requires a number of buffers to remain unused (no more than 10-15% of the total buffer space, in our experiments).

[6]The logical block size still determines the granularity of the disk transfer sizes and the buffer space (de)allocation.

5.2 Contiguity of Disk Space Allocation

Arguably, disk access efficiency would improve if the disk space corresponding to each request were allocated contiguously, requiring a single disk head movement instead of a maximum of two as incurred by stride-based allocation. We investigate this issue by measuring disk bandwidth utilization when retrieving streams allocated on a disk using different stride sizes, while still keeping the stride size larger than the stream requests in a round (as per our original constraint). The achieved stream throughput is based on the resource reservations of Section 2.4, and remains the same across different stride sizes. As was explained before, the stream strides are approximately uniformly distributed across the disk space in order to prevent disk geometry biases.

Figure 6 shows the measured bandwidth utilization of a single disk configuration when retrieving different streams. The system load was set equal to $\rho = 80\%$, and the statistics were gathered over a period of 2,000 rounds after an initial warmup of 500 rounds. One important observation from these plots is that disk utilization drops as the stride size is increased from 2 MB to 16 MB. This is not surprising, since a larger stride size reduces disk head movements, and improves disk efficiency overall.

However, Figure 6 shows that the total improvement in disk utilization does not exceed 2-3%. This percentage does not justify using larger strides (and increasing the unused storage space at the last stride of each stream). Instead, it indicates that stream

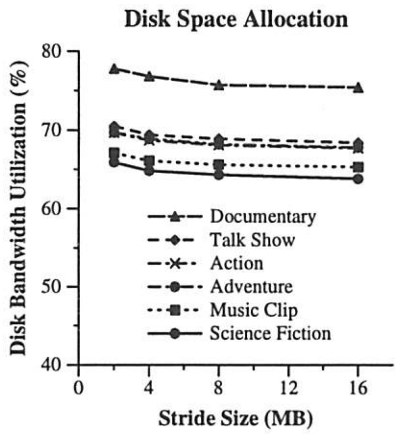


Figure 6: Increasing the stride size from 2 MB to 16 MB reduces only marginally (2-3%) the disk bandwidth utilization across different stream types. Therefore, the expected benefit from either large strides, or contiguous disk space allocation, would be limited. A single-disk configuration was used with load $\rho = 80\%$.

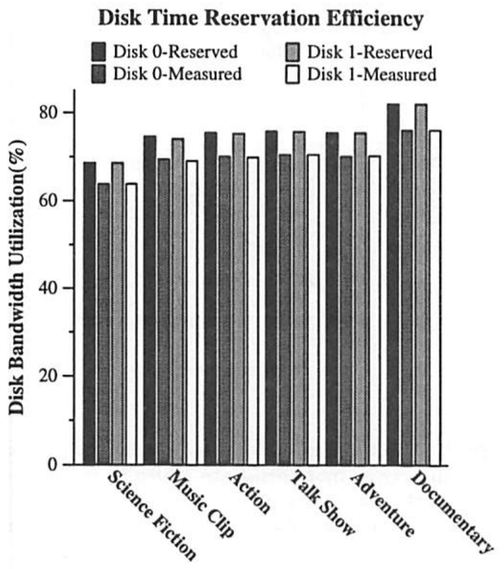


Figure 7: In a two-disk configuration with load $\rho = 80\%$, the measured disk utilization is balanced between the two disks. On each disk, the reserved disk utilization bounds relatively tightly (is only higher by about 5%) the measured disk utilization.

disk accesses are dominated by useful data transfers rather than mechanical overhead. More generally, in an environment of multiple streams striped across several disks, the expected benefit from contiguous disk space allocation would be limited. Reduction in disk actuator overhead as a result of technology advances will only make this argument stronger.

5.3 Resource Reservation Efficiency

For the following experiments we fix the buffer block size to $B_l = 16KB$ and the stride size to $B_s = 2MB$. In a system with 2 disks and 64 MB buffer memory, we compare the reserved and measured resource utilizations across different stream types.[7] We set the system load to 80%, and gather statistics for a period of 2,000 rounds after a warmup of 500 rounds. Higher loads would only lead to more rejected streams (not shown here), and would not significantly increase the system utilization. The average number of active streams in the above measurement period was roughly between 20 and 25 depending on the stream type.

Typically, the measured busy time in each round was less than or within a few milliseconds of the total disk time reserved. In only a small percentage of rounds (less than 1%) the discrepancy would be higher, and this is hard to avoid completely, due to mechanical and other kinds of unexpected overhead. However, all the discrepancies could be hidden from the client with an extra round added to the playback initiation latency. Other than that, we achieved stable prolonged system operation at high loads. The corresponding processor utilization hardly exceeded 5% on our SMP system. (We expect the processor utilization to get higher when network protocol processing is included.)

In Figure 7, we illustrate the fraction of the measurement period, during which each of the two disks was busy, and the corresponding fraction of reserved time. We notice that the load is equally balanced across the two disks. (This observation remained valid when striping streams across larger disk arrays as well, which has important scalability implications.) In addition, the reserved busy fraction does not exceed by more than 5% the corresponding measured busy time. Hence, our admission control procedure offers quality of service guarantees, without disk bandwidth underutilization.

Each of the buffer blocks allocated for a data transfer is marked busy at the beginning of the round when the disk access occurs. It is not released until its last byte is sent over the network, in some subsequent round. On the other hand, resource reservation during admission control reserves the size of each buffer for the duration of the rounds that it spans. In general, depending on the speed of the network subsystem and the network scheduling policy, we expect the measured buffer utilization to lie

[7]The resource reservations are based on the analytical estimations of Section 2.4, and are intended to be accurate predictors of the corresponding measurements in the system.

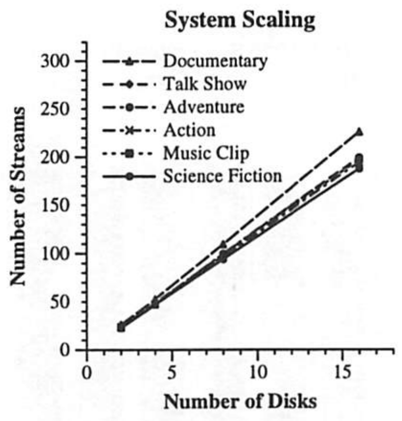


Figure 8: The number of accepted streams is projected to increase linearly as more disks are added to the system and the rest of the hardware resources increase proportionally. For these experiments, we run the system in Admission Control mode with the load at $\rho = 80\%$.

somewhere between the half and the total reserved buffer space fraction.

5.4 Disk Striping Efficiency

In order to speculate about the scaling properties of our design, we used our system in Admission Control mode, where resource reservation occurs as with our previous experiments, but without any corresponding data transfers. This allows the study of system performance scalability, when additional system resources are available in the assumed hardware configuration.

Figure 8 shows the sustained number of active streams that can be supported with increasing number of disks across different stream types. The statistics were gathered during 6,000 rounds following a warmup period of 3,000 rounds. The figure shows that the number of streams increases in proportion to the number of disks used. This is a direct consequence of the load balancing property of Variable-Grain Striping. In addition, Figure 8 shows how performance also depends on the variability of data transfers across different rounds, which is different for each stream type (Table 2). A more extensive scalability analysis of alternative disk striping policies is presented elsewhere [1].

6 Related Work

One of the better known media servers is the Tiger fault-tolerant video fileserver by Bolosky et al. It supports distributed storage of streams with constant bit rates only [3]. The Fellini storage system by Martin et al. uses a client-pull model for accessing CBR/VBR stream data and does resource reservation based on the worst case requirements of each stream [14].

In the continuous media file server proposed by Neufeld et al., detailed resource reservation is done for each round, but the study focuses on storing data of an entire stream on a single disk [19]. The Symphony multimedia file system by Shenoy et al. integrates data of different types on the same platform [24], with admission control based on peak rate assumptions. Our design, instead, is customized for storage of stream data in order to maximize efficiency.

The RIO storage system by Muntz et al. is designed to handle several different data types, including video streams [18]. The admission control is based on statistics, and the stream blocks are randomly distributed across different disks for load balancing. Instead, we use deterministic admission control in order to achieve both balanced load and high bandwidth utilization across multiple disks, as we demonstrate using actual MPEG-2 streams over SCSI disks.

An early design of distributed data striping is described by Cabrera and Long [4]. Their data striping and resource reservation policies do not take into account special requirements of variable bit rate streams, however. In addition, striped data pass through an intermediate node before being sent to the clients. We avoid such an approach for improved scalability. Keeping the metadata management of each disk separate relates in several ways to the design of the backing store server for traditional data by Birrel and Needham [2].

The idea of grouping together buffer blocks is not new either. In the design of FFS, McKusick et al. argue that chaining together kernel buffers would allow accessing contiguous blocks in a single disk transaction, and more than double the disk throughput [15]. At that time, throughput was limited by processor speed however, and changes in the device drivers were also necessary for adding this feature.

Although stride-based allocation seems similar to extent-based [16] and other allocation methods [5, 24], one basic difference is that strides have fixed size. More importantly, when a stream is retrieved, only the requested amount of data is fetched to memory and not the entire stride, which is sequentially allocated on the disk surfaces.

7 Conclusions and Future Work

We introduced the *Exedra* distributed media server architecture, and described the details of a single-node multiple-disk prototype that we have implemented. We found the separation of metadata management for each disk to greatly simplify the structure of the system, and capable of handling the case of heterogeneous disks. The dual-queue CSCAN disk scheduling method added stability to the system operation. Contiguous allocation of memory buffers simplified performance tuning, while stride-based allocation kept the disk bandwidth utilization high without adding the complexity that contiguous disk space allocation would require. Our resource reservation scheme matched relatively tightly the measured resource utilization. The sustained number of active streams increased linearly as more resources were added in the assumed configuration.

Our next step will be to extend our system to run on multiple nodes. Another important issue is tolerance of component failures through appropriate replication.

Acknowledgments

Thanks to Ben Gamsa for a useful discussion, and helpful comments on an earlier draft of this paper.

References

[1] Anastasiadis, S. V., Sevcik, K. C., and Stumm, M. Disk Striping Scalability in the Exedra Media Server. In *ACM/SPIE Multimedia Computing and Networking Conf.* (San Jose, CA, Jan. 2001). (to appear).

[2] Birrel, A. D., and Needham, R. M. A Universal File Server. *IEEE Transaction on Software Engineering* **6**, 5 (Sept. 1980), 450–453.

[3] Bolosky, W. J., Barrera, J. S., Draves, R. P., Fitzgerald, R. P., Gibson, G. A., Jones, M. B., Levi, S. P., Myhrvold, N. P., and Rashid, R. F. The Tiger Video Fileserver. In *Intl. Work. on Network and Operating System Support for Digital Audio and Video* (Zushi, Japan, Apr. 1996), pp. 97–104.

[4] Cabrera, L.-F., and Long, D. D. E. Swift: Using Distributed Disk Striping to Provide High I/O Data Rates. *Computing Systems* **4**, 4 (1991), 405–436.

[5] Chang, E., and Zakhor, A. Cost Analyses for VBR Video Servers. *IEEE Multimedia* (Wint. 1996), 56–71.

[6] Clark, T. *Designing Storage Area Networks.* Addison-Wesley, Reading, Mass., 1999.

[7] Ganger, G. R., Worthington, B. L., and Patt, Y. N. The DiskSim Simulation Environment: Version 2.0 Reference Manual. Tech. Rep. CSE-TR-358-98, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan, Dec. 1999.

[8] Garofalakis, M. N., Ioannidis, Y. E., and Ozden, B. Resource Scheduling for Composite Multimedia Objects. In *Very Large Data Bases Conf.* (New York, NY, Aug. 1998), pp. 74–85.

[9] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A Cost-Effective, High-Bandwidth Storage Architecture. In *Conf. Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 1998), pp. 92–103.

[10] Gringeri, S., Shuaib, K., Egorov, R., Lewis, A., Khasnabish, B., and Basch, B. Traffic Shaping, Bandwidth Allocation, and Quality Assessment for MPEG Video Distribution over Broadband Networks. IEEE Network, 6 (Nov/Dec 1998), 94–107.

[11] *The IBM Dictionary of Computing.* McGraw-Hill, New York, NY, 1994.

[12] Jones, M. B. The Microsoft Interactive TV System: An Experience Report. Tech. Rep. MSR-TR-97-18, Microsoft Research, 1997. ftp://ftp.research.microsoft.com/pub/tr/tr-97-18/tr-97-18.html.

[13] Lakshman, T. V., Ortega, A., and Reibman, A. R. VBR Video: Tradeoffs and Potentials. *Proceedings of the IEEE* **86**, 5 (May 1998), 952–973.

[14] Martin, C., Narayanan, P. S., Ozden, B., Rastogi, R., and Silberschatz, A. The Fellini Multimedia Storage System. In *Multimedia Information Storage and Management* (Boston, MA, 1996), S.M.Chung, Ed., Kluwer Academic Publishers.

[15] McKusick, M. K., Joy, W. N., Leffler, S., and Fabry, R. S. A Fast File System for UNIX. *ACM Transactions on Computer Systems* **2**, 3 (Aug. 1984), 181–197.

[16] McVoy, L., and Kleiman, S. R. Extent-like Performance from a Unix File System. In *USENIX Winter Technical Conference* (Dallas, TX, 1991), pp. 33–43.

[17] MPEG Software Simulation Group. *MPEG-2 Encoder/Decoder, Version 1.2*, 1996.

[18] Muntz, R., Santos, J. R., and Berson, S. A Parallel Disk Storage System for Real-Time Multimedia Applications. *International Journal of Intelligent Systems* **13**, 12 (Dec. 1998), 1137–1174.

[19] Neufeld, G., Makaroff, D., and Hutchinson, N. Design of a Variable Bit Rate Continuous Media File Server for an ATM Network. In *IS&T/SPIE Multimedia Computing and Networking Conf.* (San Jose, CA, Jan. 1996), pp. 370–380.

[20] RealNetworks, Inc. *Working with RealProducer 8 Codecs.* Seattle, WA, June 2000. Technical Blueprint.

[21] Reddy, A. L. N., and Wijayaratne, R. Techniques for improving the throughput of VBR streams. In *ACM/SPIE Multimedia Computing and Networking Conf.* (San Jose, CA, Jan. 1999), pp. 216–227.

[22] Ruemmler, C., and Wilkes, J. An Introduction to Disk Drive Modeling. *Computer* **27**, 3 (Mar. 1994), 17–28.

[23] Sen, S., Dey, J., Kurose, J., Stankovic, J., and Towsley, D. Streaming CBR transmission of VBR stored video. In *SPIE Symposium on Voice, Video and Data Communications* (Dallas, TX, Nov. 1997), pp. 26–36.

[24] Shenoy, P. J., Goyal, P., Rao, S. S., and Vin, H. M. Symphony: An Integrated Multimedia File System. In *ACM/SPIE Multimedia Computing and Networking Conf.* (San Jose, CA, Jan. 1998), pp. 124–138.

[25] Shenoy, P. J., Goyal, P., and Vin, H. M. Issues in Multimedia Server Design. *ACM Computing Surveys* **27**, 4 (Dec. 1995), 636–639.

An Architecture for Content Routing Support in the Internet

Mark Gritter, David R. Cheriton
Computer Science Department
Stanford University
{mgritter,cheriton}@dsg.stanford.edu

Abstract

The primary use of the Internet is content distribution — the delivery of web pages, audio, and video to client applications — yet the Internet was never architected for scalable content delivery. The result has been a proliferation of proprietary protocols and ad hoc mechanisms to meet growing content demand.

In this paper, we describe a content routing design based on *name-based routing* as part of an explicit Internet content layer. We claim that this content routing is a natural extension of current Internet directory and routing systems, allows efficient content location, and can be implemented to scale with the Internet.

1 Introduction

With the emergence of the World Wide Web, the primary use of the Internet is content distribution, primarily in the form of web pages, but increasingly audio and video streams as well. Some measurements indicate that 70 to 80 percent of wide-area Internet traffic is HTTP traffic. Much of the remainder consists of RealAudio streams and DNS[4, 12]. That is, almost all of the traffic in the wide area is delivery of content, and ancillary traffic to locate it. Today, millions of clients are accessing thousands of web sites on a daily basis, with relatively few popular sites supplying a large proportion of the traffic. Moreover, new popular web sites and temporarily attractive web sites can prompt the sudden arrival of a "flash crowd" of clients, often overwhelming the resources of the associated servers.

To scale content delivery to support these demands, more and more content providers and content hosting sites are replicated at multiple geographically dispersed sites around the world, either on-demand (i.e. caching) or by explicit replication. The problem is then to route client requests to a nearby replica of the content, the *content routing* problem.

In this paper, we argue that current content routing designs are unsuitable due to their closed nature and scalability limits. Section 3 describes a content routing system that forms part of an explicit Internet *content layer*, and we claim that this system provides better latency and scalability than current approaches. We support these arguments by an analysis of the scalability of name-based routing in section 4, and close with a description of related work, future goals, and conclusions.

2 The Content Routing Problem

The goal of content routing is to reduce the time needed to access content. This is accomplished by directing a client to one of many possible content servers; the particular server for each client is chosen to reduce round-trip latency, avoid congested points in the network, and prevent servers from becoming overloaded. These content servers may be complete replicas of a popular web site or web caches which retrieve content on demand.

Currently, a variety of *ad hoc* and, in some cases, proprietary mechanisms and protocols have been deployed for content routing. In the basic approach, the domain name of the desired web site or content volume is handled by a specialized name server. When the client initiates a name lookup on the DNS portion of the content URL, the request goes to this specialized name server which then returns the address of a server "near" the client, based on specialized routing, load monitoring and Internet "mapping" mechanisms. There may be multiple levels of redirection, so that the initial name lookup returns the address of a local name server which returns the actual server to be used, and the client must send out additional DNS requests.

As shown in figure 1, a client which misses in the DNS cache first incurs the round-trip time to access the DNS root server, to obtain the address of the authoritative name server for a site, e.g. `microsoft.com.`. Next the client must query this

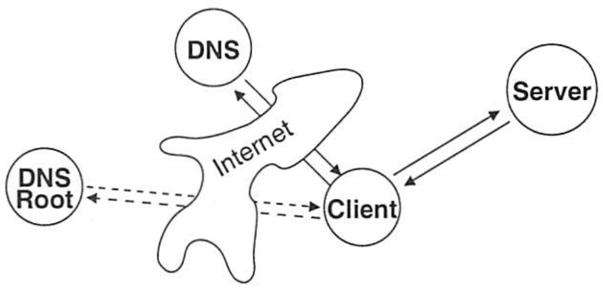


Figure 1: Conventional Content Routing

name server to receive the address of a nearby content server, incurring another round-trip time. Finally, it incurs the round-trip time to access the content on the designated server. If in this example the client is located in Turkey, the first round-trip is likely to go to Norway or London. The second round-trip may have to travel as far as Redmond, Washington, and the final might be to a content distribution site in Germany.

Thus, the conventional content routing design does not scale well because it requires the world-wide clients of a site, on a cache miss, to incur the long round-trip time to a centralized name server as part of accessing that site, from wherever the client is located. This round-trip times are purely overhead, and are potentially far higher than the round-trip times to the content server itself; this long latency becomes the dominant performance issue for clients as Internet data rates move to multiple gigabits, reducing the transfer time for content to insignificance. These name requests may also use congested portions of the network that the content delivery system is otherwise designed to avoid.

DNS-based content routing systems typically use short time-to-lives on the address records they return to a client, in order to respond quickly to changes in network conditions. This places additional demands on the DNS system, since name requests must be sent more frequently, increasing load on DNS servers. This can lead to increased latency due to server loads, as well as increased probability of a dropped packet and costly DNS timeout. As shown in [5], DNS lookup can be a significant portion of web transaction latency.

Both of these problems can be ameliorated by introducing multiple levels of redirection. Higher-level names (e.g., `m.contentdistribution.net`) specify a particular network or group of networks and have a relatively long time-to-live (30 minutes to an hour). The records identifying individual servers (such as `s12.m.contentdistribution.net`) expire in just seconds. However, this increases the amount of work a client (or a client's name server) must perform on a "higher-level" cache miss, and requires additional infrastructure. Also, such a design conflicts with the desire for high availability, since alternate location choices are unavailable to a client with cached DNS records in the event of network failure.

Conventional content routing systems may also suffer from other availability problems. A system which uses only network-level metrics does not respond to application-level failure, so a client may be continually redirected to an unresponsive web server. Designs which rely upon measurements taken from server locations may also choose servers which are not suitable from the client's perspective, due to asymmetric routing. A smaller, related, problem is that DNS requests go through intermediate name servers, so that the actual location of the client may be hidden.

Finally, content routing systems may have difficulty scaling to support multiple content provider networks and large numbers of content providers. Some content providers (such as CNN.com) serve HTML pages from a central web site but provide graphics and other high-bandwidth objects from a content delivery network; the URLs of these objects are located under the content delivery network's domain name. This has the advantage of increasing the probability of DNS cache hits, since the same server location information (`akamai.net`, for example) can be used for other sites. However, it does not help increase availability of the site's HTML content or improve latency to access it. Performing content routing on a larger set of domain names in order to improve web latency may result in lower DNS hit ratios, in addition to the costs of a larger database at a content delivery network's name servers.

Obtaining access to the network routing information needed to perform content routing may also be problematic. Content provider networks must either obtain routing information from routers near to their servers (via BGP peering or a proprietary mechanism) or else make direct network measurements. Both these schemes require aggregating network information for scalability, duplicating the existing routing functions of the network. It may also be politically infeasible to obtain the necessary information from the ISPs hosting content servers.

There is also no clear path for integrating access to multiple content delivery networks. In order to do so, a content provider would have to include an additional level of indirection to decide which CDN to direct clients to. This may be infeasible in practice (for example, if a URL-rewriting scheme is used to indicate the CDN in use), or at the very best difficult due to conflicting mechanisms and metrics. The proprietary approaches to content routing violate the basic philosophy of the Internet of using open, community-based standard protocols, imposing a closed overlay on top of the current Internet that duplicates many of the existing functions in the Internet, particularly the routing mechanisms.

3 Network-Integrated Content Routing

Our approach to the content routing problem is to view it as, literally, a *routing* problem. Clients (and users) desire connectivity not to a particular server or IP address but to some piece of content, specified by name (typically a URL). Replicated servers can be viewed as offering alternate routes to access that content, as depicted in Figure 2. That is, the client can select the path through server 1, server 2 or server 3 to reach the content, assuming each server is hosting the desired content. Thus, it is the same multi-path routing problem addressed in the current Internet in routing to a host.

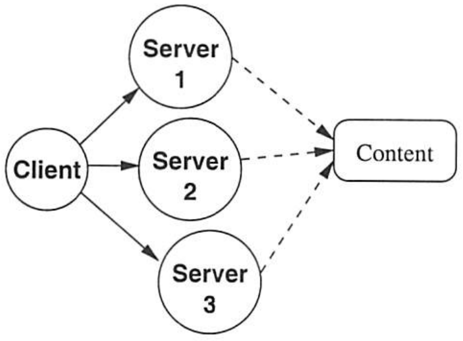


Figure 2: Content-Layer Routing

Network-integrated content routing provides support in the core of the Internet to distribute, maintain, and make use of information about content reachability. This is performed by routers which are extended to support naming. These *content routers* (CRs) act as both conventional IP routers and name servers, and participate in both IP routing and *name-based routing*. This integration forms the basis of the *content layer*. Not every router need be a content router; instead, we expect firewalls, gateways, and BGP-level routers to be augmented while the vast majority of routers are oblivious to the content layer.

3.1 Content Lookup

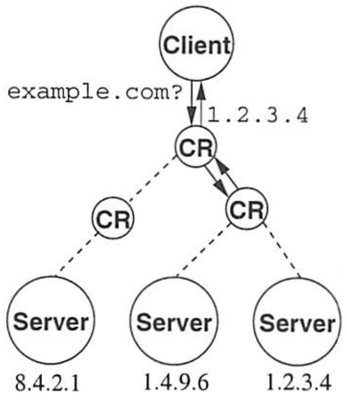


Figure 3: Internet Name Resolution Protocol

Name lookup is supported by the *Internet Name Resolution Protocol* (INRP); this protocol is reverse-compatible with DNS, using the same record types and packet format, but with different underlying semantics. Clients initiate a content request by contacting a local content router, just as they would contact a preconfigured DNS server. Their requests may include just the "server" portion of a URL, although in the long run it would be advantageous to include the entire URL.

Each content router maintains a set of name-to-next-hop mappings, just as an IP router maps address prefixes to next hops. (This name routing information is maintained using a dynamic routing protocol detailed below.) When an INRP request arrives, the desired name is looked up in the name routing table, and the next hop is chosen based on information associated with the known routes. The content router forwards the request to the next content router, and in this way the request proceeds toward the "best" content server, as shown in Figure 3. The routing information kept for a name is typically just the path of content routers to the content server, although it may be augmented with load information or metrics directly measured by a content router.

When an INRP request reaches the content router adjacent to the "best" content server, that router sends back a response message containing the address of the preferred server. This response is sent back along the same path of content routers. If no response appears, intermediate content routers can select alternate routes and retry the name lookup. A client application which is INRP-aware can also request exclusion of a non-responsive server in an INRP request.

In this fashion, client requests are routed over the best path to the desired content in the normal case, yet can recover from a failing server or out-of-date routing information. INRP thus provides an "anycast" capability at the content level, with network and client control to re-select alternatives based on direct experience with the chosen server.

Routing is done at the granularity of server names rather than full URLs (although the latter could be useful for proxies or content transformers). This decision does limit some possible caching applications, but provides little practical obstacle to web designers, since directory names from the file portion of a URL can be moved into the front of the server name (i.e., `http://foo.com/bar/index.html` can become `http://bar.foo.com/index.html`). Routing is longest-suffix match, since this can be much more efficiently performed than other possible matches on URLs.

Relaying the name lookup request across the same path as the packets are to flow ensures that naming is as available as endpoint connectivity— and that the replica selected is actually reachable. Moreover, the trust in name lookup matches the trust in delivery because both depend on the same set of network nodes. Also, the name lookup load for a path is imposed just on the routers on that path, so upgrading a router on that path for increased data capacity can also upgrade the name lookup capacity on that path. Additionally, we are exploring piggybacking connection setup these name lookups, in which case the name lookup would progress all the way to the content server itself.

3.2 Name-Based Routing

The *Name-Based Routing Protocol* (NBRP) performs routing by name with a structure similar to BGP [15]. Just as BGP distributes address prefix reachability information among autonomous systems, NBRP distributes *name suffix* reachability to content routers. Like BGP, NBRP is a distance-vector routing algorithm with path information; an NBRP routing advertisement contains the path of content routers toward a content server.

At its most basic, a BGP routing advertisement consists of an address range, a next-hop router address, and a list of the autonomous system (AS) numbers through which the advertised route will direct traffic. For example, an advertisement for Stanford's IP address range might specify `171.64/255.192` as the range, `192.41.177.8` as the next hop router, and `7170, 1` as the AS-path.

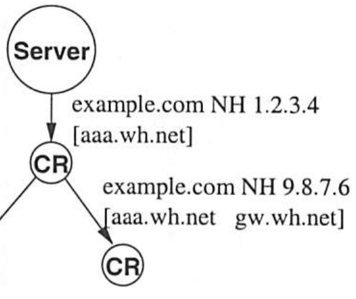


Figure 4: Name-Based Routing

As shown in figure 4, a name-based routing advertisement contains essentially the same information. The advertised content is named `example.com`, the next hop toward that content is the address of the content server or content router, and the path of routers through which the content is accessed.

Routing advertisements from content servers may also include a measure of the load at that server, specified in terms of the expected response latency. This extra attribute indicates that content which takes longer to access appears "further away" from a routing perspective, and may be treated internally by a content router as extra hops in the routing path. The distance this load information is propagated is limited to keep the number of routing updates manageable.

NBRP updates can be authenticated by cryptographic signatures, in a manner similar to Secure BGP [10]. A content server's authenticity is verified by the signature on its initial routing update; content routers receive explicit permission from this content server to advertise routes with their name added to the path list.

Content routers should apply information learned from IP routing to the content routing; if a content peer becomes unreachable then all the content available through that peer is unreachable as well. IP routing information can also be used to select among routes that appear identical at the content routing level. Finally and most importantly, IP routing policies must be consistent with content routing policies so that the decisions made at the content level are faithfully carried out by the IP forwarding level. (It is possible that existing traffic engineering schemes can be used to ensure this behavior; however, we provide some additional ideas on how the two layers can be integrated in the future work section.) Content routers may also make routing decisions based upon information obtained via measurement and mapping techniques.

3.3 Benefits

Using INRP and NBRP as described above, a client request is mapped to a nearby content server within one round-trip time to a content router near to the client, without the need to contact off-path name servers. Latency is typically dominated by round-trip time to the content server, not by content routing; cache misses require only one RTT to do a new name lookup. Moreover, by increasing the number of content routers, this property is retained even as the Internet scales to ever larger size and increasing number of clients.

By making name lookup low-latency, INRP eliminates the need to perform multiple levels of redirection in DNS. Instead, low-TTL address records can be returned at the first layer of naming, to preserve sensitivity to network conditions. (Assuming, of course, that the content routers can handle the name lookup load required, which will be addressed below.)

By making INRP and NBRP open standard Internet protocols, all ISPs, router manufacturers and content providers can participate in this content routing layer, further enhancing the cost-effective scalability to the clients.

The key issue raised by our solution is the scalability of NBRP, given it is distributing naming and load information, not just aggregatable addressing information. Ideally, we would like to completely replace the current Domain Name System by INRP and NBRP, to remove dependence on root name servers— themselves a large source of connection setup latency and scalability concerns.

4 Scaling Mechanisms for Name-Based Routing

At the global top-level domain name (GTLD) level, the domain name system is essentially flat. There is little aggregation possible with the domain name space beyond that performed at the organization level. For example, most names of the form `*.stanford.edu` appear in the same part of the network, but `stanford.edu` is not aggregatable as part of an `edu` route. So, "default-free" content routers have to know essentially all second-level domain names.

4.1 Explicit Aggregation

To handle large numbers of names which appear globally in name-based routing tables, NBRP supports combining collections of name suffixes that map to the same routing information into *routing aggregates*. For instance, we expect an ISP content router to group all of the names from its customers into a small number of aggregates. Routing updates then consist of a small number of aggregates rather than the large number of individual name entries contained in each aggregate. Load on an entire data center or network may be advertised as load on the aggregates advertised by that data center or network.

Routing aggregate advertisements contain a version number, so that a content router can detect a change in the contents of an aggregate. Aggregate contents are discovered by sending an INRP request back to the router advertising the aggregate; this request is for a "diff" between the last known version and the advertised version, so that large aggregates do not have to be resent in their entirety. Aggregate membership is relatively long-lived, compared to dynamic routing state, so that content routers can amortize the cost of learning the names in an aggregate over many routing updates.

All names in a routing aggregate are treated identically in routing calculation, thus reducing load at content routers. This is accomplished in our implementation by mirroring the indirection provided by aggregates in the routing table, as shown in figure 5. A routing table entry for a name appearing in an aggregate contains a special entry pointing to the entry for the aggregate itself. This indirect pointer is treated as the preferred route for the aggregate. Thus, when the routing information for the aggregate (`calren.net`) changes, the routing information for its constituent names (`stanford.edu` and `bekerley.edu`) is automatically updated.

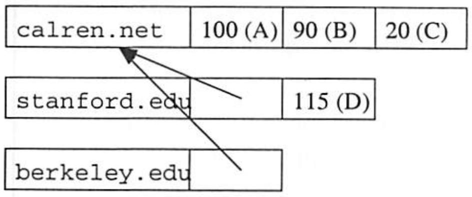


Figure 5: Routing Table with Aggregates

The number of aggregates a name belongs to is likely to be relatively small for all but the largest content providers— which can be advertised unaggre-

site_threshold	Affixes (1000s)	*aggregate_threshold* 3	5	10	20
2	1727	19.5 (6.7)	20.1 (5.6)	25.7 (4.4)	37.0 (3.4)
3	1692	14.9 (5.9)	16.1 (5.0)	20.6 (4.0)	30.1 (3.2)
10	1679	14.8 (5.9)	16.0 (5.0)	20.6 (4.0)	30.3 (3.2)

Table 1: Number of routes (and aggregates) in thousands for different site and aggregate threshold values.

gated. So, even though this design requires a linear search through the entire list of routes for aggregated names, we expect that this cost will be relatively small. (It would eliminate much of the benefit of aggregation to re-sort all such lists on a routing change.)

Fine-grain information such as server load is hidden by the aggregate; regions of the network where the aggregate is advertised but individual members are not must base their decision on just the aggregate. This is similar to the way BGP routing provides coarse-grained address range information and does not indicate whether particular hosts or subnets are up or down. In fact, the situation is improved by INRP's request-response nature: unlike datagram delivery, a name lookup can pursue alternate routes if a route proves inaccurate— at the cost of additional latency.

To evaluate the expected performance of aggregation if applied in the current Internet, we processed a comprehensive list of address-to-name mappings in the Domain Name System[6] and BGP table dumps from the MAE-East exchange point[7] by the following algorithm, making the assumption that name-based routing structure will roughly correspond to current BGP autonomous system boundaries:

1. Each address range from the BGP table is matched with the DNS zones represented. (If fewer than *site_threshold* hosts in a range belong to an existing zone, they are removed from the table completely and assumed to be handled with the redirection mechanism described below in section 4.2)
2. Names whose associated routing information is made redundant by a superzone are also removed.
3. Aggregates are created for any set of names larger than *aggregate_threshold* that have identical routing information (i.e., all known routes were identical, not just the preferred route.)

The resulting aggregates, although incomplete, provide an estimate of those expected to be generated by name-based routers.

One representative set of results is shown in Table 1. The original BGP table had 68,200 routing table entries. Aggressive aggregation (site threshold of 10 and aggregate threshold of 3) results in a table with 1,679,000 entries, but only 14,800 advertised names (including 5,900 aggregates). Thus, the number of routes is actually smaller than in the original IP routing table. Even relaxed values of the model parameters result in a routing "back-end" size comparable with the original BGP table. Higher-level aggregation may be able to reduce this yet further without resorting to renumbering or renaming.

The number of routing entries is comparable with the best possible achieved under IP routing. BGP does have a limited mechanism for aggregation: a single route update may include several address prefixes. It is not clear the extent to which BGP software makes use of this to optimize update calculations: there is no requirement that advertisements keep these address prefixes together, and the address ranges must appear separately in the IP routing table. Aggregating all all identical address prefixes would result in 11,800 routing table entries for the original MAE-East table.

And additional challenge NBRP addresses is addition of a new names, which is much more common than addition of new BGP prefixes; this name information must propagate to all default-free content routers. This is far smaller than the rate of normal routing updates, since addition of new names is done on human time scales; even the addition of 10 million new globally routed suffixes per year results in just 19 updates per minute. Some new names are added to multiple locations or duplicated to handle flash crowds, increasing the rate of routing table change. However, the number of routing updates seen by an individual router on name addition is limited by the number of direct peers. To put this in perspective, a backbone router may receive more than 2,000 routing updates per minute; new naming information is dwarfed by the normal rate of topological change.

Also, the actual level of routing updates necessary for new names can be lower in some cases because changes to aggregates can be "batched" to reflect many new names with one update.

The cost of distributing the contents of routing aggregates is acceptable as well, even at Internet scales. The aggregates obtained from the analysis described above show a heavy-tailed distribution; the mean number of names per aggregate is 304, while the median is 24. The average size of the domain names in these aggregates is 16 bytes, although these statistics could be somewhat different if the content routing system was used more aggressively to do redirection on finer granularity. Even an aggregate of 50,000 entries requires only 782 kilobytes to be sent initially, estimating 16 bytes for each name suffix. Later updates can be sent as deltas to the known aggregate, since aggregates can be kept in permanent storage. The long-term bandwidth consumed by aggregate updates (across as single peering connection) can be estimated as

$$\text{number of names} \cdot \frac{2 \cdot (3 \cdot \text{average name size} + 170)}{\text{average name lifetime}}$$

The estimate represents the cost of sending a query (including IP and TCP headers) for a change in aggregate membership and getting the response; this transaction occurs twice since it happens when the name enters and leaves an aggregate. The packet sizes were measured by our implementation and assume one TCP packet per query. Even assuming a relatively short time of 1 day for the average time a name suffix appears in an aggregate, a database of 30 million names (with average size 16) requires 1.2 Mbit/second data transfer. When compared to the 10-gigabit expected capacity of backbone links, this number represents only 0.01 percent of the available bandwidth. Moreover, all aggregate routing update traffic takes place only between two immediate NBRP peers.

4.2 Redirection

Not all hosts with names in a given suffix are connected to the network globally advertising that name suffix. For example, there may be hosts with `stanford.edu` names scattered throughout the Internet, even though most Stanford names are located together. It does not seem feasible to advertise all of these names globally. Such hosts could simply be assigned fixed addresses by the content router operated by Stanford. However, we see some benefit in allowing local flexibility in address assignment without updating a remote server— particularly in situations where network address translation is being used, or in mobile networks.

INRP provides a redirection mechanism for finding isolated names not advertised in the the name-based routing system. Such names have records indicating their actual topological location in the Internet in terms of a more well-known name. When a request is answered with a redirection record, the client (or the first-hop content router acting on its behalf) restarts the query using the proper name. For example, if the host `gritter.stanford.edu` is located at Berkeley, a name lookup might return a redirection to `guest32.berkeley.edu`. This redirection mechanism trades fate-sharing and name lookup latency for decreased routing state; economic factors may well determine what names appear in routing tables and which are found through redirection. That is, an ISP may charge per name it places in the routing table, so an organization weighs the cost-benefit of having a name handled by the ISP versus incurring the redirection cost on a name.

This secondary mechanism is really only needed when using NBRP to replace all DNS usage; for content routing, name-based routing tables contain only site and content volume names rather than host and network interface names.

5 Implementation and Analysis

Our prototype content router has been implemented in C++. The name routing table is implemented as a hash trie, allowing longest-suffix matching to be performed in time linear with name length. (For most names in our sample and experiments this is simply two hash table lookups.) The following measurements were taken on a 600 MHz Pentium III system running Linux 2.2.12.

5.1 Content-Layer Overhead

We measured an overhead of 0.5 milliseconds (total for both request and response) for going through a single hop of the content routing layer, on a name routing table of 5 million entries stored entirely in memory. The 5 million names were randomly generated second-level domain names, with 80% in `.com` and 10% in each of `.org` and `.net`; a uniform distribution of name lengths between 3 and 17 was used. These names were divided into aggregates of 15,000.

Measurements done on the 1.7 million-name database from our aggregation experiment show no significant difference in overhead. Profiling information shows that most of this time is spent doing packet processing; measurements on the actual routing table show that route lookup takes as little as 6 microseconds. Our implementation can easily sustain a throughput of 650 requests/second without any degradation of response time, and peak throughput of 1600 requests/second.

The total amount of memory used by the content router for a 5 million entry table was 344MB, while that on a similarly generated 100,000 entry (but unaggregated) table was 20MB. This leads to an estimate of 69 bytes per routing table entry. We can extrapolate that a 30-million entry database would require nearly 2GB of memory; while large, this is not an infeasible amount of DRAM, costing only about $4000. (It is worth noting that name lookups which must go to the DNS root *already* encounter a database lookup of approximately this size.)

5.2 Improved Performance with INRP

The improved performance provided by INRP is illustrated by considering access times to content servers through Akamai versus our proposed content routing. An additional example shows the benefit of using INRP rather than contacting root name servers.

A conventional name lookup of `a388.g.akamai.net` from Stanford returns the addresses of two content servers which are located 6.6 ms round-trip-time away.

At the next level, the name servers for `akamai.net` are located throughout the Internet, with an average round-trip times ranging from 12 ms to 93 ms. Overall, this set of name servers has a mean response time of 65 ms and median of 83 ms, ignoring dropped requests.

Using INRP, the same request would go through about 5 content servers (at least one per intervening network), so we will estimate 3 ms extra round-trip time. The direct path to the content servers would then require approximately 10 ms for the name request. A similar example for a miss at the root name servers is carried out in Table 2 for `www.cisco.com`.

As the latency measurements in Table 2 indicate, INRP reduces average request latency in these examples by 86 to 95 percent and also eliminates the variability in latency, providing more predictable performance.

5.3 Name-Based Routing Performance

We measured the routing throughput of our prototype implementation on a local network using a random routing update traffic generator. A single machine was the source of all routing traffic; an instrumented content router was configured to advertise its preferred routes to a variable number of peered content routers, all connected by a 100Mbit LAN. To maximally exercise the content router, the generated traffic consisted of previously unknown routes and changes to aggregate membership, so that all routing updates were propagated to all configured peers. The routing preference function used was minimal; with more complicated routing polices, the cost of calculating route preferences dominates, so the results presented below should be considered upper bounds on the performance of this particular implementation.

Site	Server Prefix	request latency minimum	request latency mean
Akamai	`akamai.net`	12 ms	65 ms
	`g.akamai.net`	7 ms	7 ms
	Total	19 ms	72 ms
	INRP (5 hops)	10 ms	10 ms (-86%)
Cisco	`com`	9 ms	101 ms
	`cisco.com`	4 ms	40 ms
	Total	13 ms	141 ms
	INRP (5 hops)	7 ms	7 ms (-95%)

Table 2: Example name request round-trip times on cache miss (measured from Stanford) for `a388.g.akamai.net` and `www.cisco.com`.

Figure 6 shows the routing throughput for each number of peers. The "no aggregates" data set represents routing updates advertising individual names. Throughput gracefully declines with the number of routing peers, from a maximum of 1050 updates per second with one peer, to 370 updates per second with six peers.

For "1 percent update", 1% of the routing advertisements consisted of a one-name change to an aggregate. As the figure shows, this reduces throughput by approximately 5%, due to the extra queries needed to obtain the new name and the file system accesses to store the new aggregate contents. Increasing the update size to 20 names showed only a 1-2% additional reduction in throughput, indicating there is some benefit to batching aggregate membership changes. Higher proportions of aggregate changes to normal routing updates result in further reductions in throughput; an experiment where all

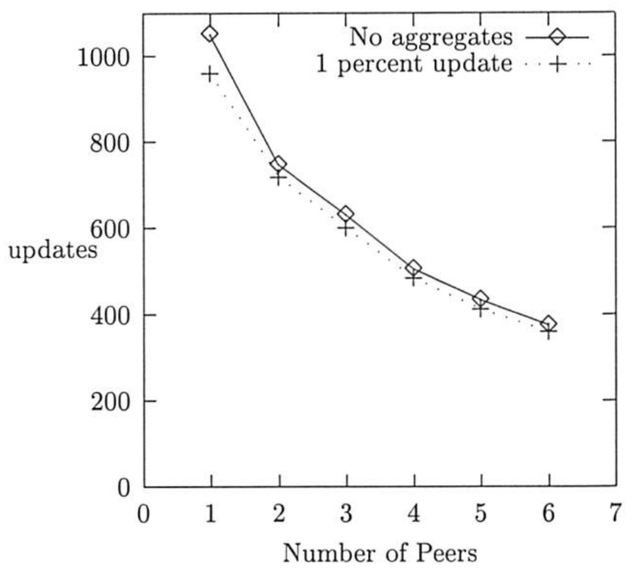


Figure 6: Name-based Routing Performance (updates/sec)

routing updates were aggregate changes resulted in only 87 updates/second.

6 Deployment

The content layer has a simple deployment path, based on user need and on an ISP's motivation to provide a better web experience to customers and superior service to colocated service providers. INRP and NBRP can initially be implemented in ISP name servers, which fail over to normal DNS behavior for unrecognized names. INRP provides a way for ISPs to quickly direct their customers to colocated servers, eliminating any need for name requests to leave their network. NBRP is not strictly needed, but may prove a convenient way to advertise new content to an ISP's name servers.

This initial deployment requires no changes to end hosts and no change to the basic IPv4 routers and switches constituting the infrastructure of the leaf and backbone networks. It only requires the deployment of content routers, which can be implemented on top of existing hardware using packet filtering and redirection techniques. In particular, hosts still use conventional DNS lookup to get an address, but benefit from reduced dependence on distant root name servers and lower latency to access local content servers. However, some customers may be running their own name servers and avoiding the use of ISP DNS servers, and thus see no benefit until they reconfigure.

The overhead of doing content routing in this manner is very small, since the ISP's name server already does DNS packet handling. Only the cost of accessing the name-based routing table would be wasted on names not in the content routing system— much less than the 0.5 ms described above.

ISPs who already peer at the IP routing level are motivated to peer at the content routing level to provide their customers faster access to nearby content servers— and increase the benefit of placing content servers in their network. As demand grows, additional content routers can be placed to handle the increased usage without user-visible changes. As the content routing topology evolves to more closely matches IP routing topology, the content routing system can make more accurate decisions.

7 Related Work

The original Internet directory service was supplied by a "hosts.txt" file that listed all hosts in the Internet. As the Internet grew, this approach was replaced by DNS [13] in 1985. Subsequent work on "network directories" such as X.500 attempts to support naming of other types of objects such as mailboxes and users, and providing more flexible ways of specifying identification, such as lists of attributes. We choose instead to restrict the entities being named (and the namespace) in order to improve scalability.

Content routing builds on the *decentralized naming* approach advocated from experience in the V distributed system [3], That is, as in V, names are the primary identification of objects, hosts and content volumes in this case, and each server implements the naming for the objects it implements. There is no centralized name repository.

Current wide-area content routing depends on HTTP- or DNS-level redirection, and is generally done on the "server side". For instance, Cisco's Distributed Director (DD) redirects a name lookup from the main site to a replica site closer to requesting client address, based on responses from a set of participating routers running an agent protocol, supporting DD. Unfortunately, the client incurs the response time penalty of accessing this main site DD before being directed to the closer site. Proprietary schemes by Akamai, Sightpath, Arrowpoint and others appear to work similarly. These proprietary content distribution networks can be centrally monitored and managed, unlike name-based routing. This may lead to better understanding of network performance; however, CDNs still rely upon the existing IP routing framework for content delivery, so the amount of benefit to be gained from a proprietary overlay network is limited. Additionally, we believe future work on advanced routing designs and improved network management is applicable to name-based routing as well as IP routing. (As research and experience with BGP has shown [11], wide-area routing is neither easy to understand nor easily tuned.)

Smart Server Selection [14], in contrast, is a "client-side" approach to content routing. An authoritative name server for a content volume returns *all* available addresses for replicas of the content, and the client (or the client's name server) interacts with a nearby router to obtain routing metrics for this set of addresses and chooses the nearest one. This requires cooperation from the router in the form of a request protocol, adding an extra step to the name lookup process. Smart Server Selection also does not address the problem of name lookup latency.

Similarly, some DNS servers measure round-trip times to known name servers in order to choose the lowest-latency server, especially at the root level. Although this can improve the performance of name lookups by lowering the mean lookup latency, it only helps at one level of a cache miss. Further, such name servers are ignorant of network conditions and thus may experience several timeouts before switching their preference to an alternate server.

Intentional naming [1] integrates name resolution and message delivery, offering application-layer anycast and multicast similar to our proposed content layer. The Intentional Naming System offers attribute-based naming, a much richer form of content addressing than URLs or domain names. However, INS is not designed to provide global reachability information, and the attribute-based naming is less scalable than the a hierarchal namespace provided by URLs. INS's "late binding", where every message packet contains a name, is too expensive to use for content distribution; our proposed architecture corresponds to INS's "early binding", where names are resolved to addresses before content is exchanged.

Much work has been done on distributed caching schemes; one design very similar in spirit to our content-layer routing is "adaptive web caching" [16]. In this system, caches exchange information about which web pages they currently hold (in order to eliminate the need for "cache probing") and maintain "URL routing tables". Our design does not offer routing on the granularity of URL prefixes, as adaptive web caching does, but offers a more comprehensive solution intended to replace current naming systems.

Network-level anycast designs such as GIA [8] attempt to solve server location problems at the IP level. The semantics of an anycast IP address are to deliver a packet destined for that address to the "best" of an available pool of servers. GIA does not incorporate application-level metrics, so anycast packets may be routed to an unresponsive server without providing any recourse for the client. Also, unless a client is statically configured with all needed anycast addresses, it must still use a directory to determine the address to use.

8 Future Directions

The motivation for a "content layer" approach came as part of the TRIAD[2] project. TRIAD is a new, NAT-friendly Internet architecture which seeks to reduce dependency on addresses by promoting names as transport-layer endpoints. In a TRIAD Internet, all large-scale routing would occur at the naming level. We believe this approach is ultimately more scalable and deployable than attempts to solve problems (such as mobility, multihoming, anycasting, and wide-area addressing) at the network level.

Two features of TRIAD enhance the content routing architecture. TRIAD provides extended addressing via the *Wide-Area Relay Addressing Protocol* (WRAP), which provides loose-source routing among multiple address realms. WRAP addresses can be used to specify a path through the network, ensuring that the route selected by the content routing layer is the path actually used by data packets. TRIAD also integrates TCP connection setup into INRP name lookup; by sending TCP connection initiation information inside an INRP request, the latency for web transactions can be lowered yet further. Thus, the full TRIAD architecture integrates naming, routing, and connection setup into a single framework.

INRP allows proxies and web caches to intercept content requests based on URL. We have not implemented or fully explored this design, but it appears to be a promising way to provide "semi-transparent" proxies, which would require no explicit configuration at the client, but would be used by the client as a content request's TCP connection endpoint.

Finally, the integration of naming and routing allows *feedback-based routing.* Conventional IP routing schemes have few ways to tell if the routes they select actually deliver packets to the intended destination. Content routers, however, can track the responses they receive to forwarded queries, allowing them to make better decisions and react more quickly to routing problems than conventional routers. For example, if content servers send back load information in INRP responses, then content routers can obtain up-to-date load information on heavily used sites without placing this load information into routing updates.

The most current version of this paper can be found at [9].

9 Conclusions

Current content routing solutions will not scale to handle increasing global demands for content. Conventional content routing distributes content delivery but does not effectively distribute content discovery. Further, the proprietary nature of most content routing designs in use today makes them undesirable for global use and are in conflict with the Internet open standard philosophy.

The content layer — integrated naming and routing — provides a mechanism for large-scale content routing that addresses these issues. By pushing naming information out into the network, content routers allow fast location of nearby content replicas; in essence, content routers provide the same service for naming that CDNs do for the content itself.

We developed NBRP to distribute names in this fashion and INRP to perform efficient lookup on this distributed integrated named-based routing system. Our results indicate that client name lookup is then faster and far less variable.

The content layer can be easily deployed to provide immediate benefits to ISPs and their customers. Our implementation, and the networking community's experience with BGP, give confidence that name-based routing can scale at least to the demands of content routing for popular content. We anticipate that additional research and experience will demonstrate the feasibility of using name-based routing for all Internet naming.

10 Acknowledgments

The TRIAD project was supported by the US Defense Advanced Research Projects Agency (DARPA) under contract number MDA972-99-C-0024. This paper greatly benefited from the guidance of our shepherd, Steven Gribble, and the comments of the USITS reviewers; thanks also to Vincent Laviano and Dan Li for their feedback.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley, "The design and implementation of an intentional naming system", Proc. 17th ACM SOSP, Dec. 1999.

[2] David Cheriton and Mark Gritter, "TRIAD: A New Next-Generation Internet Architecture", `http://www.dsg.stanford.edu/triad`, July 2000.

[3] D. R. Cheriton and T.P. Mann, "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance", ACM TOCS, May 1989.

[4] k claffy, G. Miller, and K. Thompson, "The nature of the beast: recent traffic measurements from an Internet backbone", INET 98.

[5] E. Cohen and H. Kaplan, "Prefetching the means for document transfer: A new approach for reducing Web latency", INFOCOM 2000.

[6] Internet Domain Survey, July 1999, `http://www.isc.org/ds/`.

[7] Internet Performance Measurement and Analysis project, `http://www.merit.edu/ipma/`.

[8] Dina Katabi and John Wroclawsi, "A Framework for Scalable Global IP-Anycast (GIA)", SIGCOMM 2000.

[9] Mark Gritter and David R. Cheriton, "An Architecture for Content Routing Support in the Internet", `http://www.dsg.stanford.edu/papers/contentrouting/`, 2001.

[10] S. Kent, C. Lynn, and K. Seo, "Secure Border Gateway Protocol (S-BGP)", IEEE Journal on Selected Areas in Communication, 1999.

[11] Craig Labovitz, G. Robert Malan, and Farnam Jahanian, "Internet Routing Instability", ACM SIGCOMM Conference, September 1997.

[12] Sean McCreary and kc claffy, "Trends in Wide Area IP Traffic Patterns", ITC Specialist Seminar on IP Traffic Modeling, Measurement and Management, September 2000.

[13] P. Mockapetris, "Domain Names – Concepts and Facilities", RFC 882, November 1983 (obsoleted by RFC 1034 and 1035).

[14] W. Tang, F. Du, M. W. Mutka, L. Ni, and A. Esfahanian, "Supporting Global Replicated Services by a Routing-Metric-Aware DNS", Proceedings of the 2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000), June 2000.

[15] Y. Rekhter, T. Li (editors), "A Border Gateway Protocol 4 (BGP-4)", RFC 1771, March 1995.

[16] Lixia Zhang, Scott Michel, Khoi Nguyen, Adam Rosenstein, Sally Floyd, and Van Jacobson, "Adaptive Web Caching: Towards a New Global Caching Architecture", 3rd International WWW Caching Workshop, June 1998.

ALMI: An Application Level Multicast Infrastructure

Dimitrios Pendarakis
Tellium Optical Network Systems
dpendarakis@tellium.com

Sherlia Shi
Department of Computer Science
Washington University in St. Louis
sherlia@arl.wustl.edu

Dinesh Verma
IBM T.J. Watson Research Center
dverma@watson.ibm.com

Marcel Waldvogel
Department of Computer Science
Washington University in St. Louis
mwa@arl.wustl.edu

Abstract

The IP multicast model allows scalable and efficient multi-party communication, particularly for groups of large size. However, deployment of IP multicast requires substantial infrastructure modifications and is hampered by a host of unresolved open problems. To circumvent this situation, we have designed and implemented ALMI, an application level group communication middleware, which allows accelerated application deployment and simplified network configuration, without the need of network infrastructure support. ALMI is tailored toward support of multicast groups of relatively small size (several 10s of members) with many to many semantics. Session participants are connected via a virtual multicast tree, which consists of unicast connections between end hosts and is formed as a minimum spanning tree (MST) using application-specific performance metric. Using simulation, we show that the performance penalties, introduced by this shift of multicast to end systems, is a relatively small increase in traffic load and that ALMI multicast trees approach the efficiency of IP multicast trees. We have also implemented ALMI as a Java based middleware package and performed experiments over the Internet. Experimental results show that ALMI is able to cope with network dynamics and keep the multicast tree efficient.

1 Introduction

This work is motivated by the need to support group communication among a small group of hosts without relying on the IP multicast model. Traditional IPmulticast, as defined by IGMP and related standards, provides an excellent solution to the communication needs of multicast groups with a large number of members. However, it requires fairly elaborate control support from network devices, such as IP routers, in particular membership management (IGMP) and multicast routing protocols. Since IP routers maintains separate routing state for each multicast group, the model is relatively less scalable with respect to the number of concurrently active multicast groups. Widespread deployment of IGMP and routing protocols requires substantial infrastructure modifications, and complex modifications to IP routers' software. Some of the issues associated with IP multicast, e.g. end-to-end reliability, flow and congestion control schemes, offer significant challenges for which no clear solutions have emerged thus far.

There are a large number of applications whose requirements are substantially different from the design point of IP multicast. Such applications include video-conferencing, multi-party games, private chat rooms, web cache replication and database/directory replication. These applications usually contain a small number of group members, and the groups (e.g. multi-party games) are often created and destroyed relatively dynamically. The number of such groups that are concurrently active can be fairly large. For a large number of such small and sparse groups, the benefits of IP multicast in terms of bandwidth efficiency and scalability are quite often outweighed by the control complexity associated with group set-up and maintenance.

Due to the increasing number of such applications, and a lack of ubiquitous deployment of IP multicast in all IP-based networks, there has been renewed interest in multicast protocols that can be supported without relying on the IP multicast infrastructure. Some of the work has been motivated by applications like Internet TV, which are single source applications with a very large group size. These schemes, which include *Simple Multicast* [14], *Express* [11] and very recently, *Source-Specific Multicast* [10], offer multicast routing schemes which solve some of the problems of their traditional IP multicast counterparts, such as address allocation and access control. Nevertheless, all these solutions require substantial changes to the network infrastructure and their adoption by the network community and deployment in the Internet is yet to be seen.

In order to meet the requirements of emerging applications, we need a solution for multi-sender multicast communication which scales for a large number of communication groups with small number of members, and does not depend on multicast support in the routers. In this paper, we propose an *application level multicast infrastructure* that addresses these concerns. This solution provides a multicast middleware which is implemented above the socket layer. Application level multicast offers accelerated deployment, simplified configuration and better access control at the cost of additional (albeit small) traffic load in the network. Since application level multicast is implemented in the user space, it allows more flexibility in customizing some aspects, e.g. data transcoding, error recovery, flow control, scheduling, differentiated message handling or security, on an application-specific basis.

In our scheme, participants of a multicast session are connected via a *virtual multicast tree*, i.e. a tree that consists of unicast connections between *end hosts*. The tree is formed as a *Minimum Spanning Tree (MST)*, where the cost of each link is an application specific metric. The implementation we describe in subsequent sections uses the round-trip application level delay between group members as this cost metric. However, a plug-in architecture enables this metric to be changed easily by applications. In this paper, we present the architecture of the multicast middleware we have developed, a Java based implementation, and the results of some performance experiments conducted over a local area network as well as over the Internet. We have called this Java based package, ALMI for *Application Level Multicast Infrastructure.*

The rest of the paper is organized as follows. We first present an overview of our architecture, including the operation of control and data planes in section 2, followed by a design of application specific components in section 3. Sections 4 and 5 present simulation analysis and experimental evaluation of ALMI, respectively. Section 6 describes related work; we conclude in section 7.

2 Architecture and Operations

In this section, we describe the communication channels provided by ALMI and its related protocol operations for both controller and group members. We also describe operations related to multicast tree generation and criteria of tree updates and its stability issues. One of the advantage gained in ALMI is its value-added application specific components. To simplify explanation, we defer our design of these functionalities to next section.

2.1 Overview of ALMI Architecture

An ALMI session consists of a session controller and multiple session members. Session controller is a program instance, located at a place that is easily accessible by all members. It may be co-located with one of the session members, typically the session initializer, or it could reside on a special purpose server or a multicast proxy installed within a corporate or an ISP network. Session members are organized into a multicast tree. A link in the multicast tree (solid line) represents a unicast connection between two members. Session data is disseminated along this multicast tree, while control messages are unicast between each member and the controller. The multicast tree is a shared-tree amongst members with bi-directional links. In order to avoid loops, two members incident on a link receive a designation of parent and child. This parent-child relation only distinguishes the two member for reasons we will explain later in this section; it does not indicate direction of data flow.

The session controller handles member registration and maintains the multicast tree. In order to achieve the latter, the controller performs two important functions:

- It ensures connectivity of the multicast tree when members join and/or leave the session and when network or host failures occur.
- It ensures the efficiency of the multicast tree by periodically calculating a minimum spanning tree based on the measurement updates received from all members. To collect measurements the controller essentially instructs each member to monitor a set of other members.

A session member receives and sends data as it would in an IP multicast session; in addition, it also forwards data to designated adjacent neighbors. Data eventually reaches all session members through this relaying process in a cooperative fashion. Besides forwarding data on the data plane, a session member also monitors the performance of unicast paths to and from a subset of other session members. This is achieved by periodically sending probes to these members and measuring an application level performance metric; in the current implementation the roundtrip response delay. Delay measurements are then reported to the controller and serve as the costs used to calculate a Minimum Spanning Tree.

ALMI takes the *centralized control* approach to maintain tree consistency and efficiency. This design choice is made for better reliability and reduced overhead during a change of membership or a recovery from node(i.e. end system) failure. On the other hand, the session controller manifests itself only in the control path, and does not obstruct high data rate transmissions among session members. We believe this centralized approach is adequate and efficient for a large range of multicast applications. However, a centralized controller architecture has obvious implications in control plane reliability and fault tolerance. Clearly, a single controller would constitute a single point of failure for all control operations related to the group. Two points should be made in this respect. First, the centralized session controller could be augmented with multiple *back-up controllers*, operating in "stand-by" mode, with addresses which are well known to all session members. In this case the "stand-by" controllers periodically receive state from the primary controller, which would include recent measurements, tree topology and current membership information. Second, even in the event that no control operation is possible, the existing ALMI tree, and hence data path, will remain unaffected and will continue operation until a membership change or a critical failure occurs. Therefore a transient controller (or its network) failure can be tolerated. In summary, we believe the benefit of simplicity offered by the centralized controller approach far outweigh any negative implications from the fault tolerance perspective.

2.2 Control Plane Operation

ALMI relies on a *control protocol* for communication between session controller and session members. This protocol handles tasks related to membership management, performance monitoring and routing. ALMI uses a common packet format to carry both data and control packets, shown in Figure 1.

The content of this packet header is rather straight forward. *Session ID* and *Source ID* are generated by controller and guaranteed to be collision free. The *flag* field in the header defines various types of operation messages, including:

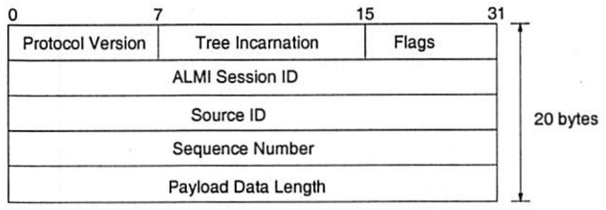


Figure 1: ALMI Packet Header Format

- Registration messages addressed from hosts to the controller. When a host joins a session, the controller returns a list of peering points from which the member should accept connection requests and the parent to which the new member is to initiate a connection.

- Connection request and acknowledgment between parent and child. This message exchanges parent and child data port numbers, which are locally bound with either TCP accept sockets or with UDP sockets. Members use these ports to initialize future data connection.

- Performance monitoring messages reported from members to session controller, such as pairwise delay measurements between group members. Each update message includes a list of <*current neighbor ID, delay measurement*> pairs and is recorded by controller for the use of calculating the minimum spanning tree.

- Distribution tree messages, generated by the controller, are used to inform members of their peering points in the data distribution tree. This message informs members of their new parent and children's ID. It typically occurs after detection of network or system errors, or after a tree transition.

- Neighbor monitoring update messages, which are sent by the controller to members to inform them a new list of neighbors they need to monitor. This message is triggered if the controller detects the number of current monitoring pairs has dropped below a threshold due to accumulated network errors. Or it is triggered due to the unsatisfaction of the current state of the multicast tree.

- Departure messages, are sent from group members to the controller and their current parent and children. If a child member receives such a message from its parent, it needs to contact the controller again to rejoin the group.

The *Tree Incarnation* field is to prevent loops and partitions in the multicast tree. Since a session multicast tree is calculated centrally by the controller, assuming correct controller operation, a loop free topology will always be generated. However, since tree update messages are independently disseminated to all members, there is always a possibility that some messages might be lost or received out-of-order by different groups members. In addition members might act on update messages with varying delay. All of these events could result in loops and/or tree partition. In order to avoid these transient phenomena, the controller assigns a monotonically increasing version number to each newly generated multicast tree. To avoid loops, a source generating packets includes its latest tree incarnation in the packet header. In order to guarantee tree consistency and at the mean time ensure delivery of most packets, each ALMI node maintains a small cache of recent multicast tree incarnations. Thus, an ALMI node simultaneously keeps state about multiple trees, each with the corresponding list of adjacent nodes. The number of cache entries is configurable. When receiving a packet with tree version contained in the cache, the receiving node forwards it across the interfaces corresponding to this tree version. Packets with tree versions not contained in the cache are discarded. On the other hand, if a member receives a data packet with a newer tree version, it detects that its information is not up to date and therefore re-registers itself with the controller to receive the new tree information.

2.3 Member Operation

One of the first tasks a session member has to perform is to locate the session controller. It is assumed that initially, the session ID, the controller's address and port number are communicated or announced to members through online or offline schemes, such as a URL, a directory service or an email message. A session member is identified by its network address and port number, the combination of which will subsequently be referred to as the member's *address*. Members register by sending a *JOIN* message to the session controller. A member accepted to the group receives from the controller its *member ID*, as well as the ID and address of its parent. The member then sends a *GRAFT* message to its parent and in response obtains the data ports on which it receives and sends data.

Data distribution along the multicast session tree occurs on a hop by hop fashion. Depending on the application, data transfer between two adjacent members can be reliable or unreliable by deploying TCP (e.g. data replication services) or UDP (e.g. stream-based applications), respectively. There are clear advantages in being able to use existing, widely deployed protocols: first, it reduces system administration and configuration cost; and second, use of TCP and its associated congestion mechanism offers hop-by-hop reliability and provides compatibility in bandwidth sharing with regular flows. We stress that the last property is rather convenient since multicast congestion control is an extremely hard problem especially for its deployment viability. Additionally, applying TCP on a hop-by-hop basis implicitly creates back pressure for the source to slow down, resulting in end-to-end, albeit simplistic, congestion management.

When TCP is used, a connection has to be established between two adjacent nodes with one end initiating and the other end accepting the connection. Therefore, ALMI controller assigns parent and child labels to two adjacent nodes: a TCP connection is always initialized in the direction from a child to the parent. The parent-child relationship is also used in monitoring connectivity; if a child detects failure of the connection to its parent, it considers itself disconnected from the graph and sends a *REJOIN* message to the controller. On the other hand, if the parent detects a child connection failure, it simply closes the connection. This relationship does not indicate directions of data flows, however, once the multicast tree is formed, each member forwards data to all adjacent members, including all children and the parent, except the one on which data is received.

As part of the evolving tree dynamics, a session member might be required to switch to a new parent. Such an event can be initiated by either the controller ("push") or the member ("pull"). In the former case, the controller instructs the member to switch to a new parent because a substantially better MST has been computed. In the latter case, the member detects through the monitoring process that its parent is not responding or receives a *LEAVE* message from the parent. It then issues a *REJOIN* message to the controller, repeating the steps as when joining an ALMI group. In both cases, determination of a new parent is made by the controller.

2.4 Multicast Tree Generation and Update

We now turn to the computation of the ALMI distribution tree. A session multicast tree is formed as a virtual *Minimum Spanning Tree* that connects all members. The minimum spanning tree calculation is performed at the session controller and results are communicated to all members in the form of a *(parent, children)* list. Link costs are representative of application specific performance metric which is computed by members in a distributed fashion and reported to the controller periodically. In our current implementation, we uses roundtrip delay, measured at ALMI layer, as the metric because latency is important to most applications and is also relatively easy to monitor. However, some applications may find other metrics such as available link bandwidth, more useful and better suited to match its performance measure. As an example, a bandwidth intensive application may prefer a high bandwidth, high delay link to a low delay, low bandwidth link to carry its traffic. Design and development of these type of tools to obtain more sophisticated measurements helps ALMI to provide more flexible services and these tools can be easily plugged in as a module to ALMI. Nevertheless such instrumentation in a wide area network is non-trivial and it is beyond the scope of this paper to discuss these mechanisms. In the rest of this paper, we will simply use delay as the default performance metric.

Neighbor monitoring graph

In order to obtain monitoring results, ALMI connects all group members into a monitoring graph. Members send ping messages to measure round trip delay to its neighbors in the graph. For small groups, it is possible to create a mash and have $O(n^2)$ message exchanges to compute the best multicast tree. However, as group size grows, it becomes unscalable to have large number of message exchanges since the monitoring process is periodic and continuous through the whole multicast session. To reduce control overhead, we limit the degree of each node in the graph, i.e. the number of neighbors monitored by a member, to be constant so as to reduce the number of message exchanges to $O(n)$. The consequent spanner graph results in sub-optimal multicast tree since it does not have a complete view of all possible paths and its set of edges may not be a super set of all edges in MST. Such sub-optimality is reduced, however, by occasionally purging the currently known bad edges from the graph and updating it with edges currently not in the graph. Over time, the graph converges to include all edges in the optimal degree-bounded spanning tree. Likewise, in a dynamic environment, the graph updates to trace the better set of edges and to produce a more favorable multicast tree.

Multicast tree and its stability

Once members start to report monitoring results to their session controller, ALMI is able to improve the multicast tree from its initial random tree.[1] As described above, an ALMI multicast tree is a degree-bounded optimal spanning tree. Since most end hosts tend to be on access links rather than at network core, it is desirable to confine the number of packet copies traversing through access links to be small, i.e a small degree bound. On the other hand, if servers use ALMI to construct a multicast session and they have access to high speed network, the degree bound can be correspondingly configured higher.

A more crucial issue is how to achieve stability of the multicast tree since a change of tree is associated with operational cost such as *GRAFT*, *GRAFT ACK* and re-initiation of the data connection. More over, data packet may be lost or duplicated during a tree transition, and recovery process can be expensive for it incurs additional delay and data buffering at the application. Therefore, our goal of improving the performance of multicast tree is only on a long term basis and any potential path oscillations are prevented. The controller calculates the overall performance gain of the new multicast tree and switches tree only if the overall gain exceeds a threshold. Both the frequency and threshold of switching tree are user configurable parameters.

3 Design of Application Specific Components in ALMI

Previous sections presented the architecture of control and data planes in ALMI. One of the advantages in ALMI is its ease of deploying value-added services for applications, such as end-to-end reliability, data integrity and authentication, and quality of service. A complete design of building blocks to fulfill these requirements is outside the scope of this paper. This section discusses briefly design points in supporting some of these components and in particular, we present our design and protocols for a reliable data distribution service which we have recently implemented.

3.1 End to End Data Reliability

Content distribution applications typically require data consistency and reliability. TCP has success-

[1]By default, the set of neighbors in the multicast tree is a subset of neighbors in the monitoring graph, so a re-computation can only result in performance improvement.

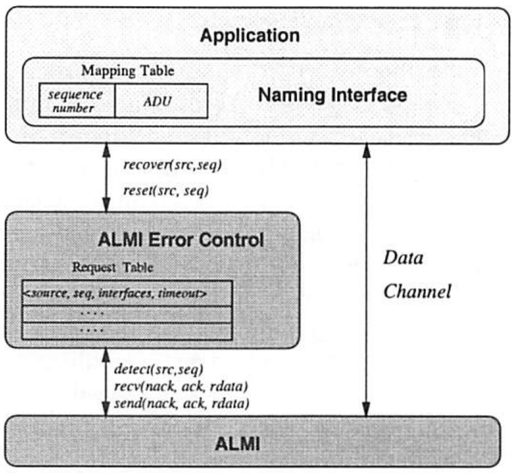


Figure 2: ALMI Naming and Error Control

fully satisfied these requirements for unicast connectivity; a TCP-equivalent reliable transport protocol for multicast communication has been the subject of active research in recent years [12]. In an ALMI multicast group, the end-to-end reliability problem still exists; however, the cause of the problems differs greatly from that over IP multicast. In ALMI, unicast TCP connections provide data reliability on a hop-by-hop basis, which implies that packet losses due to network congestion and transmission errors are eliminated. Instead, the main reason for packet losses in ALMI are due to multicast tree transitions, transient network link failures, or node failures.

In ALMI, implosion and exposure control happens naturally, it efficiently aggregate requests and retransmit data without the need for router support or knowledge of session topology. Upon loss detection, a session member sends a request onto the interface where data is received from. Requests are then aggregated at each hop so that only one of them escapes the loss subtree. When applications can buffer data or regenerate data from disk, retransmission can happen locally. In this case, the node above the lossy link will retransmit data to the requesting subtree. Otherwise, when upstream node has reset its application naming states(explained below) and can no longer retransmit data locally, a *NODATA* packet is sent back to the requestor, i.e. the head of the loss subtree. The requestor then initiates an *out-of-band connection* directly to the source, and subsequent request and retransmit are conducted over this out-of-band connection. In both local and out-of-band retransmission, upon receiving retransmitted packets, requestor forwards them to downstream requestors. The out-of-band connection is torn down after fulfilling the request. The choice of out-of-band request versus relaying request and retransmissions hop-by-hop is due to ALMI's loss characteristics: they are infrequent but usually happen in bulk. Typically, once a node loses its connection, it takes about 3 round trip time to re-connect to the multicast tree and detect packet losses. Although relaying request all the way up to the source can sometime aggregate more independent loss requests at higher up the tree, it adds per-hop processing and transmission delay for each request and retransmission packet, and also disrupts the normal data distribution process. On the contrary, an out-of-band connection separates data distribution from retransmissions and have much less processing delay.

Additionally, ALMI also deploys *ACKs* to synchronize data reception states at members. This is necessary for applications that require total reliability but have limited buffer space. Before resetting their buffers, members need to ensure all packets in buffer are correctly received by all members. An *ACK* is a list of <source, sequence number> pairs, where sequence number is the highest contiguous sequence number received locally from a data source. Initiated from leaf nodes, *ACKs* are sent upstream towards the root. At each intermediate node, once a member received *ACKs* from all its children, it forwards upstream an *ACK* containing the minimum of sequence numbers for each source. When the *ACK* reaches root, it is multicasted back downstream and reset every nodes' state to their common minimum. A member is then free to clear up all packet buffers with sequence number less than the minimum. The frequency of the *ACK* process depends on both the data rate and the smallest buffer space at a member application.

3.2 Data Naming

An important question related to error recovery is that of *data naming*. Applications and ALMI require a commonly understood naming convention so that they can communicate which data is requested. Since losses in ALMI group are more likely to occur in batches over dispersed time intervals rather than isolated packets on regular time intervals, sequence numbers as used by TCP, are insufficient to specify a member's data reception state and could hinder a members' ability to request and retransmit data efficiently. Furthermore, an application may decide to ignore certain packets, for example, packets containing out-of-date information, and only recover others. A data naming component is thus more desirable since it allows flexibility in tailoring application reliability semantics.

In ALMI's data naming interface, an application can specify the mapping between its *application data units* and ALMI packet sequence numbers. An *ADU* is solely defined by application protocol, for example, for some database applications, it can be an object ID; or for a ftp application, a tuple containing <file name, offset, length>. Other more sophisticated mechanisms such as hierarchical data naming schemes [15, 5] can be incorporated as well, to achieve better flexibility and efficiency.

3.3 Other Components

There are many other functionalities that could be incorporated into ALMI, such as delay constraints for real-time sessions, access control for private multicast sessions and etc. In ALMI, an application delay bounds can be achieved by constraining the diameter of the computer MST tree. Similarly, the multicast tree can be computed with constraints on the degree of session members, in order to achieve better load balancing. Regarding access control, the session controller is naturally capable of controlling which members are allowed to join; furthermore, the controller can act as a key distribution center, distributing symmetric keys to encrypt the data, as well as certificates and signed public keys that should be used for data authentication. We are currently underway adding these components to ALMI.

4 Simulation Analysis of ALMI Multicast Tree Efficiency

While ALMI achieves group communication without relying on network layer multicast support and reduces the control load associated with group setup and maintainance, it is bound to exhibit lower transmission efficiency since nodes on the distribution tree have to be ALMI capable and, thus currently confined to end hosts. Moreover, packet processing and forwarding at the application layer typically incurs higher delay when compared to router processing at the IP layer. In this section we investigate the extent of these ALMI performance constraints by conducting experiments which compare ALMI to IP multicast. Results obtained provide insight onto the trade-offs associated with ALMI and allow us to decide the applicability of ALMI for specific applications and deployment settings.

We examine the relative cost of an ALMI tree to those of source-rooted shortest path multicast trees as well the cost of a mesh of unicast connections which would have to be used in the absence of any multicast support. Trees are generated and costs computed over a set of random graphs with a variable number of multicast group members. The algorithms for generating random graphs are similar to those in [19], where a connected graph is generated with a specified edge connectivity probability.

In comparing the cost of an ALMI multicast tree to that of source-rooted shortest path multicast trees we note that since ALMI constructs a shared multicast tree, the cost of distributing data is the same independently of the location of the sender(s). However, this property does not hold for source-rooted trees, in which data originating at different nodes will traverse paths of differing cost to reach all group members. Therefore, to achieve a meaningful comparison, the cost of an ALMI multicast tree is compared with the average cost of all shortest path trees rooted at each group member.

As mentioned in Section 2, ALMI provides a mechanism to further reduce control traffic load by allowing members to collect delay measurements to only a subset of other group members. Obviously, performing the MST calculation on a (connected) subgraph results in a sub-optimal ALMI distribution tree. In this section, we analyze quantitatively the impact of this mechanism in terms of how much it increases the cost of the actual ALMI multicast tree. The cost of an ALMI tree is defined to be the sum of delays on each link of the shared multicast tree; all link delays are assumed to be symmetric.

Figures 3 and 4 depict multicast tree cost in a random graph and a transit-stub graph, respectively. Each data point is derived by averaging over the results of 10 graphs. Random graphs in Figure 3 consist of 500 nodes with an average node degree of 5, and transit-stub graphs in Figure 4 consist of about 6000 nodes, with an average node degree of 3. More details about the formation of transit-stub graphs can be found in [19]. Link costs are uniformly distributed in the interval $[0, 1]$.

In both figures, the x-axis of the graph on the left depicts multicast group size; groups of variable size are formed by selecting a random subset of network nodes as group members. It is assumed that every network node can be co-located with a host. The graphs on the left plot the average cost of all source-rooted trees, one for each multicast group node, the ALMI MST cost and the cost of a mesh of $O(n^2)$ unicast connections among all group members. We also compute the cost of an ALMI multicast tree calculated from incomplete information, denoted as "ALMI sparse MST". This tree corresponds to the case where every ALMI node monitors the delay to just 10% of the total number of group nodes.

We first concentrate on the results depicted in

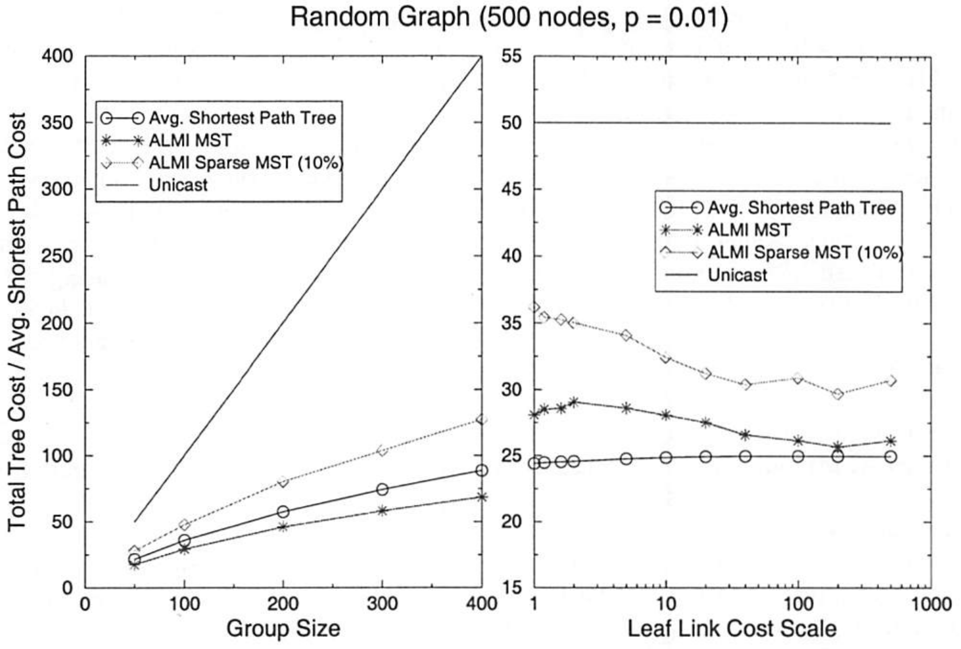


Figure 3: Cost Comparison of ALMI MST and Shortest Path Tree in Random Graph

the left graphs of figures 3 and 4. It is interesting to observe that for the random graph, at all group sizes the ALMI MST cost is smaller than the average source-based tree cost. This is essentially due to the fact that an ALMI multicast tree is an MST tree; optimal source based trees are computed based on information local to each node and, therefore, are not globally optimal. On the other hand, in a transit-stub graph, the ALMI multicast tree is about 20% more expensive. This difference is due to the distinct characteristics of the two types of graphs. Since an ALMI multicast tree consists of a collection of unicast paths between hosts, some network links will be typically traversed multiple times. In a transit-stub graph, since hosts reside in stub networks, the links between transit domains and stub domains will most certainly be traversed multiple times, whereas in the random graph topology, since hosts are co-located with network nodes and uniformly distributed throughout the graph, the number of such links are fewer, hence lowering the cost of the ALMI multicast tree. Finally, as expected, the ALMI sparse MST has a higher total cost since it is derived using a subset of link metrics. Still, the cost difference in all cases is within 50%, which could be considered a reasonable price to pay for a 90% reduction in performance monitoring traffic.

Thus far, we have assumed that all network links have equal cost and that hosts are co-located with network nodes; in other words host are attached to the network with zero cost. In practice, however, this assumption might not be accurate; typically *"last mile"* links have lower bandwidth and thus result in higher delays and MST costs. Higher "last mile" costs could adversely impact ALMI, since all data flows in and out of non-leaf nodes in the ALMI tree at least twice and hence, the cost of link connecting hosts to a network aggregation point will contribute more to the total tree cost. In the right side graphs of Figure 3 and 4, we plot tree costs against the cost of the "last-mile" links. We include the same comparisons; ALMI MST, ALMI "sparse MST", average of all shortest path trees and meshed unicast connections. In this simulation, multicast group size is fixed to 50 and the "last-mile" link cost is uniformly distributed between 0 and *scale*, shown on the x-axis.

The results demonstrate that, even for a moderate group size of 50 members, the benefit of ALMI over pure unicast is still significant, reducing tree cost to only half. Furthermore, it is observed that as the cost of "last-mile" links increases, ALMI multicast tree cost decreases and approaches the cost of the average shortest path tree. This is due to the fact that MST calculation results in a tree which tends to prefer inclusion of low-cost links. This is similar to the behavior that would be observed if servers were deployed in the network to help relay data to

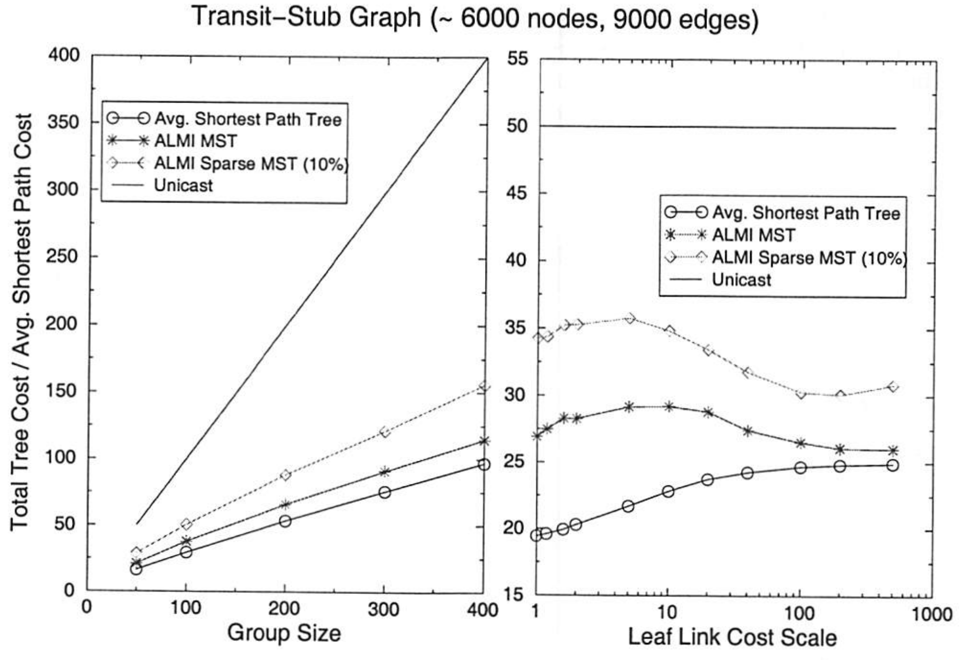


Figure 4: Cost Comparison of ALMI MST and Shortest Path Tree in Transit-Stub Graph

other parts of the network. Overall, the simulation clearly shows the advantage of an ALMI multicast tree over $O(n^2)$ unicast connections. The fact that ALMI is almost as efficient as the shortest path trees even in the presence of incomplete measurements, argues that it is a rather attractive solution for many multicast applications.

In this simulation, we have focused on comparison of ALMI multicast tree with source-rooted shortest path trees. Compliment to SPTs, shared multicast tree, as constructed from CBT [1] and PIM-SM [7] optimizes the total cost of the multicast tree. Although it is known that finding the optimal center for the multicast group is an NP-complete problem, there are heuristic placement strategies to select one of the group member or network node to be the core. In [18], it shows that a resulting shared multicast tree from a feasible heuristic method has an average cost of 95% of the cost of shortest path tree for a varied number of group sizes, average node degree and different node distributions. Therefore, we infer that the cost difference between ALMI multicast tree and CBT or PIM-SM will be comparably small as well.

5 Experimental Evaluation of ALMI

We have implemented ALMI as a Java-based middleware package using JDK1.2 [17]. In the next two experiment sets, we evaluate the performance of an actual operational group of ALMI nodes over either a WAN or a LAN. These two scenarios have fundamental differences; in a LAN environment most of the delay between two ALMI nodes is due to host processing while over a wide area network, delay is mostly due to transmission, propagation and queuing delay over the network.

5.1 Experiment Over WAN

Over a wide area network, ALMI has to cope with the dynamics of network paths, such as distortion of delay measurements and transient link failures. ALMI needs to prevent the multicast tree from diverging from an efficient construction. To demonstrate that ALMI is able to achieve a cost-efficient tree, we have conducted experiments over 9 sites scattered in both US and Europe.

The experiment was run as follows. We started ALMI at all 9 sites and configured the ALMI controller to re-calculate the multicast tree every 5 minutes. Simultaneously, we run traceroute from each site to every other site periodically, every 5 minutes. The output from traceroute provides us with a benchmark of the network delay experienced between nodes during our experiment. We then compare the total delay of an MST computed from the traceroute measurements to that of the ALMI multicast tree computed by the ALMI controller. For this experiment, we used the traceroute measured delay as the ALMI tree link cost in order to achieve a fair comparison. In other words, the comparison reflects only the difference of tree composition, excluding the distortion caused by delay measurements at the application level.

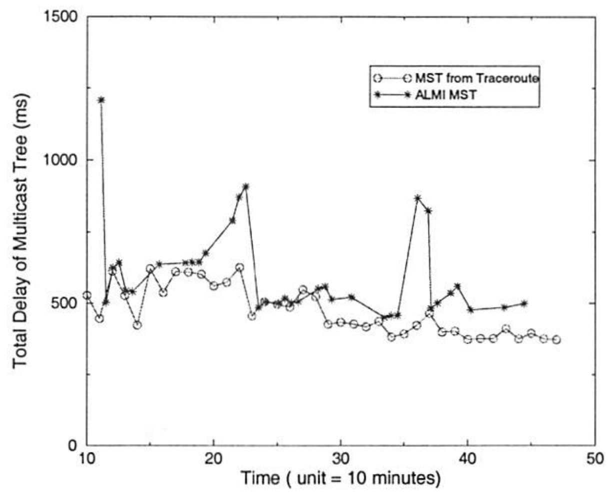


Figure 5: Evaluation of ALMI MST in WAN Test

Figure 5 shows the result of a six hour test run of a single multicast session. Initially, the cost of ALMI multicast tree is very high, since the ALMI controller does not have *a priori* topological knowledge about group members and randomly connects members to each other at the beginning of the session. However, the ALMI tree cost was quickly brought down at the next re-calculation of the tree and stays close to the real MST cost, as the controller periodically gathers measurement reports from group members and updates the ALMI MST. There are two spikes in the ALMI MST, at time units 22 and 36 respectively. Analyzing the traces, we found that both points are caused by transient network failures. In the first case, one of a pair of two nodes, who are very close to each other, detects the other end as unreachable and connects to a much higher cost neighbor. In the second case, one node experiences temporary network failure and is timed out at the controller. The network recovers after approximately 15 minutes and the node re-joins the group but is randomly assigned a new parent. The presence of a new member, either at the session beginning or during the session, always introduces sub-optimality of the tree since they are randomly connected to the rest of the ALMI multicast tree. A more intelligent controller may be able to use one of the Internet services such as in [9, 16, 13] to estimate the topological information of a new member and initialize its connection more efficiently. We conclude from this experiment that ALMI is able to use application perceived delay to construct an efficient multicast distribution tree in a highly dynamic network environment.

5.2 Experiment Over LAN

In this experiment, we test a scenario where network bandwidth is higher than what end-systems can consume, and test the forwarding processing delay caused by ALMI processing. We used a Sun Ultra-1 attached to a 10Mb/s Ethernet network as a source sending data to several Pentium III - class PCs connected over a 100 Mb/s LAN. We vary the number of intermediate data relaying hops and measure the throughput at the last hop. In this experiment, we use TCP connection between nodes and confine the controller to connect members as a chain in order to capture the effect of ALMI member node forwarding.

From Table 1, we observe that the throughput achieved in all cases remains stable regardless of the number of intermediate hops. This shows that ALMI processing delay does not increase with the higher number of data relaying hops. From a scalability point of view, this means that the overall TCP throughput achieved in a session is decided by the slowest network path or intermediate hop, but is not affected by the aggregation of bottlenecks if there are multiple. On the other hand, if we look at Table 1 vertically, we see that the processing delay associated with each packet is relatively high, especially for small size packets. This is due to the fact that the Java virtual machine is still comparably slow even in the presence of JIT. However, we believe this gap will be reduced in the near future with the advances of better compilers and faster CPUs.

6 Related Work

Challenging the conventional wisdom of IP multicast, ALMI explores an alternative architecture to apply multicast paradigm in the current Internet. There are two closely related projects emerging independently at the same time which have very similar objectives as ALMI does. Yallcast [8], aims to extend the Internet multicast architecture and defines a set of protocol for host-based content distribution either through tunneled unicast connections or IP multicast wherever available. It uses a *rendezvous host* to bootstrap group members into the multicast tree. The functionality of the *rendezvous host* is similar to ALMI's group controller, it is only used to inform new members about several current members in the tree and is not con-

packet size	**Zero Hop** (KB/S)	**One Hop** (KB/S)	**Two Hops** (KB/S)
64	156.83	154.83	153.994
128	278.57	209.98	190.56
256	489.26	439.19	422.69
512	657.81	642.83	609.13
1024	752.47	732.85	769.74
2048	800.55	797.33	788.63
4096	813.84	813.18	836.82

Table 1: Experiment of ALMI forwarding delay in end systems

nected to the multicast data paths. Yallcast creates a shared multicast tree using a distributed routing protocol. It also maintains a mesh topology among group members to ensure that the multicast group is not partitioned. Overall, Yallcast envisions the deployment of IP multicast into small and disjunct network islands and provides a rudimentary architecture for global multicast. In contrast to Yallcast, Endsystem Multicast [4] is more similar to ALMI in aiming towards small and sparse group communication applications. In Endsystem Multicast, group members are self-organized into multicast trees using a DVMRP [6] like routing protocol and creates source-based multicast tress. It require members to periodically broadcast refresh messages to keep the multicast tree partition free. A companion protocol of Endsystem Multicast is called Narada, which focuses on optimizing the efficiency of the overlay, in terms of delay bounds, based on end-to-end measurements. Both Yallcast and Endsystem Multicast are still in their initial evaluation stage and at this point, we are not aware of any performance reports. Although, Yallcast and Endsystem Multicast have their end goals align with those of ALMI, the tree construction algorithms are very different in all three protocols. Both Yallcast and Endsystem Multicast try to leverage the existing multicast routing protocols and re-apply them at the application level. However, we argue that one of the fundamental complexities comes with IP multicast is its complication in routing protocols. Although, at the application level, such complexity can be greatly reduced, due to the number of nodes involved is much fewer than the number of routers all over the Internet, a fully distributed algorithm may still cause excessive control overheads and incur reliability problems, which are the same problems as existed in current multicast routing protocols. A centralized control protocol as the one in ALMI, with careful design of redundancy, can simplify the matter greatly and provides a more reliable mechanism to prevent tree partitions and routing loops.

There are other relevant projects that also deploy multicast at the application level, with more emphasis on each specific applications. RMX [3] is a project that installs multicast proxies to connect islands of IP multicast with co-located homogeneous receivers. Besides relaying data, an RMX proxy also adapts to the heterogeneous environment using detailed application knowledge. For example, an RMX proxy can act as a transcoder to accommodate the low bandwidth receivers. The tree configuration among RMX proxies are static right now and there is no self-configuration and adaptation aspects of the multicast overlay as of this writing. AMRoute [2] is a protocol for host-based multicast over mobile wireless networks. It assumes the existence of an underlying broadcast mechanism for configuration purposes. AMRoute continuously creates a mesh of bidirectional tunnels between a pair of group members. Additionally, each multicast group has a *core node* which is responsible for the initial signaling and tree creation. The AMRoute core uses a source routing approach, where source is the core node itself, and selects a subset of the available virtual mesh links to form a multicast distribution tree. The core can also migrate dynamically according to group membership and network connectivity. Both of these projects bear similarities to ALMI, yet ALMI is defined as a more general infrastructure for a wide range of applications rather than for a specific application or environment.

7 Conclusions and Future Work

This paper presented ALMI, an application level multicast infrastructure, that has been designed and built to provide a solution for multi-sender multicast communication which scales to a large number of communication groups with small number of members, and does not depend on multicast support at the IP layer. This solution provides a multicast middleware which is implemented above the sockets layer. Application level multicast offers accelerated deployment, simplified configuration and better access control at the cost of small additional traffic load in the network. Simulation results, along with initial experimentation results indicate that the performance tradeoff is quite small and that ALMI multicast trees are close to the efficiency of IP multicast trees. Since application level multicast is implemented in the user space, it allows more flexibility in customizing some application related modules, e.g. data transcoding, error recovery, flow control, scheduling, differentiated message handling and security.

We plan to extend this work in multiple ways. We are enhancing the performance evaluation work to include experiments with a larger number of nodes, as well as integrating with real life applications so that, besides control, data performance characteristics can be studied in detail. In addition, we plan to implement and study in more detail application specific modules such as end-to-end reliability, naming and security. In terms of speeding the performance of our middleware, we will explore options of moving parts of the forwarding functionality to an OS kernel and defining an interface between ALMI and the OS specific parts.

8 Acknowledgment

Our sincere thanks to the following people for their generosity of letting us use their computing resources for conducting our experiments: Ernst Biersack of Eurecom, France; Andreas Festag of TU Berlin, Germany; Aiguo Fei of University of California, Los Angeles; Ralph Keller of ETH Zurich; Manamohan Mysore of University of California, San Diego; Christos Papadopoulos of University of Southern California; Injong Rhee of North Carolina State University; Ellen Zegura and Peter Wan of Georgia Institute of Technology. Without their help, this work would never have been completed.

References

[1] A. Ballardie. Core Based Trees (CBT version 2) Multicast Routing - Protocol Specification. RFC 2189, 1997.

[2] E. Bommaiah, L Mingyan, A. Mcauley, and R. Talpade. Amroute: Adhoc multicast routing protocol. Internet Draft, August 1998.

[3] Y. Chawathe, S. McCanne, and E. Brewer. RMX: Reliable Multicast in Heterogeneous Networks. In *Proc. IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[4] Y. Chu, S. Rao, and H. Zhang. A Case For EndSystem Multicast. In *Proceedings of ACM Sigmetrics*, Santa Clara, CA, June 2000.

[5] J. Crowcroft, Z. Wang, A. Gosh, and C. Diot. RMFP: A Reliable Multicast Framing Protocol. Internet Draft, March 1997.

[6] Distance Vector Multicast Routing Protocol - DVMRP. RFC 1812.

[7] D. Estrin, V. Jacobson, D. Farinacci, L. Wei, S. Deering, M. Handley, D. Thaler, C. Liu, Sharma P., and A. Helmy. Protocol Independent Multicast-Sparse Mode (PIM-SM): Motivation and Architecture. Internet Engineering Task Force, August 1998.

[8] P. Francis. Yallcast: Extending the Internet Multicast Architecture. http://www.yallcast.com, September 1999.

[9] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. Gryniewicz, and Y. Jin. An Architecture for a Global Internet Host Distance Estimation Service. In *Proc. IEEE INFOCOM*, 1999.

[10] H. Holbrook and B. Cain. Source Specific Multicast. IETF draft, draft-holbrook-ssm-00.txt, March 2000.

[11] H. Holbrook and D. Cheriton. IP Multicast Channels: EXPRESS Support for Large-scale Single Source Applications. In *Proc. ACM SIGCOMM*, Boston, MA, September 1999.

[12] Katia Obraczka. Multicast Transport Mechanisms: A Survey and Taxonomy. In *IEEE Communications Magazine*, January 1998.

[13] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An Architecture for Large-Scale Internet Measurement. *IEEE Communications*, pages 48–54, August 1998.

[14] R. Perlman, C. Lee, A. Ballardie, J. Crowcroft, Z. Wang, T. Maufer, C. Diot, J. Thoo, and M. Green. Simple Multicast: A Design for Simple, Low-Overhead Multicast. IETF draft, draft-perlman-simple-multicast-03.txt, October 1999.

[15] S. Raman and S. McCanne. Scalable Data Naming for Application Level Framing in Reliable Multicast. In *Proc. ACM Multimedia '98*, Bristol, UK, September 1998.

[16] S. Seshan, M. Stemm, and R. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proc 1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, Monterey, CA, December 1997.

[17] Java^{TM} 2 Platform. http://www.javasoft.com.

[18] L. Wei and D. Estrin. The Trade-offs of Multicast Trees and Algorithms. In *Proc. of IC3N*, 1994.

[19] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. IEEE INFOCOM*, San Francisco, CA, 1996.

CSP: A Novel System Architecture for Scalable Internet and Communication Services

Hemal V. Shah, Dave B. Minturn, Annie Foong, Gary L. McAlpine, Rajesh S. Madukkarumukumana, and Greg J. Regnier
Enterprise Architecture Lab
Intel Corporation
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124
{hemal.shah, dave.b.minturn, annie.foong, gary.l.mcalpine, rajesh.sankaran, greg.j.regnier}@intel.com

Abstract

The boundary between the network edge and the front-end servers of the data center is blurring. Appliance vendors are flooding the market with new capabilities, while switch/router vendors scramble to add these services to their traditional transport services. The result of this competition is a set of ad-hoc technologies and capabilities to provide services at the network edge. This paper describes the *Comm Services Platform* (CSP); a system architecture for this new 'communication services tier' of the data center. CSP enumerates a set of architectural components to provide scalable communication services built from standard building blocks that utilize emerging server, I/O and network technologies. The building blocks of CSP include a System Area Network, the Virtual Interface Architecture, programmable network processors, and standard high-density servers.

1. Introduction

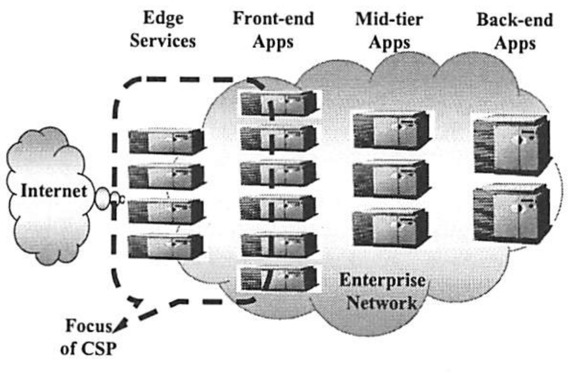


Figure 1: The N-Tier Datacenter

The three-tier data center model is well known. The first tier consists of front-end servers that provide web, messaging and various other services to clients on a network, the middle tier handles transaction processing and implements the data center business logic, while the back-end consists of databases that hold persistent state. We see a fourth tier emerging in this model between the network and the front-end servers. This 'communication services tier' provides an ever-richer set of functions. These services operate on network traffic at and above the network layer, with well-known examples such as load balancing (at multiple levels), security (firewall and SSL), caching and others. These services intend to increase the responsiveness and throughput of the data center by distributing and offloading these functions from the server farm at the front end of the data center.

This paper describes a novel system architecture for a scalable, high-performance communication services tier based on emerging server and network technologies that are, or will become, standard building blocks. These technologies include System Area Network (SAN) technology, the Virtual Interface Architecture [20], programmable network processors, and standard high-density servers. We also describe a new core service, the SAN Proxy service that offloads and decouples the Transmission Control Protocol /Internet Protocol (TCP/IP) processing from the front-end servers, thus improving the performance of all of the services built on top of TCP/IP.

The organization of the rest of the paper is as follows. Section 2 discusses observations and motivations behind our research. Section 3 provides an overview of the CSP system architecture. Solutions and techniques developed in CSP are presented in Section 4. Initial evaluation of CSP using a system prototype and simulation results are provided in Section 5.

Finally, Section 6 summarizes our findings thus far and outlines possible future work.

2. Observations and Motivations

Evolution at the data center network edge has been driven primarily by demands to make web services more scalable, efficient and available. The most common solution applied to the scaling problem is to construct systems comprised of a large number of front-end servers (web server farms) behind load or content distributors. These systems typically use commodity servers for web services and specialized appliances, commonly termed "web-switches", as distributors of web service requests. The connectivity between web-switches and front-end servers is predominately on a commodity LAN fabric, a.k.a. Ethernet, with Internet Protocol (IP) running over it.

Over the past few years, there has been an extensive amount of research on the functionality, design, and usage of web-switches. This research, along with a proliferation of commercial products [28, 29, 30, 34, 35], has mostly focused on the following areas:

- Web request distribution based on server load, content locality, cache affinity, or client session affinity [1, 14, 26, 28].
- Optimizations of server TCP connection management [4, 24, 34].
- Mechanisms for inter-connecting client and server TCP connections (TCP-splicing) [3, 12, 19].
- Offloading server CPU intensive operations, such as secure sockets layer (SSL) operations.

Each of these approaches has been based on optimizing and distributing web transactions by transparently intercepting and modifying the Hyper Text Transfer Protocol (HTTP)/TCP/IP packet dialog between the client and the server. All of these approaches operate under the premise that the web switch or appliance device must be interconnected to the front end servers, or other web appliances, using the standard TCP transport, even though the client end to end TCP connections are terminated at the web-switch. As a result, the web-switch is required to maintain two separate TCP sessions, one to the client and another one to the server. Typically, each client connection is paired with an associated server connection. Some web-switches have added an optimization that allows multiple client sessions to be multiplexed on a set of persistent web-switch to server TCP connections [34]. This reduces the server TCP connection setup and teardown overheads but adds the complexity of session management and TCP data buffering to the web-switch.

We have observed an evolution of web-switches from stateless load distributors into intelligent devices that have control over both the client and server sessions. When co-located in a data center environment, along with front-end servers and other web appliances, they function more as peers in distributed Internet end-point applications environment. Given this observation, we propose that the data center web deployments are good candidates for the application of SAN technologies. SAN technologies, specifically the Virtual Interface (VI) Architecture and emerging Infiniband™ Architecture [9], were developed to eliminate inefficiencies incurred when using general-purpose network interfaces with transports such as TCP, for high-speed inter-process communications and I/O.

We propose that a systematic approach can be applied using SAN building blocks to solve the following issues with today's data center systems:

1. The high software overheads of using general-purpose network interfaces significantly reduce the capacity of the front-end servers. Operating system related inefficiencies incurred in network protocol processing, such as user/kernel transitions, data copies, software multiplexing, and reliability semantics reduce both CPU efficiency and overall network throughput.

2. The knowledge gained by TCP/IP packet processing in a given component is not shared with other components in the system leading to excessive redundant processing and poor integration. For example, a TCP session establishment and subsequent session identification are performed on both the web-switch and the front-end server.

3. General-purpose network protocols are not suitable for low-latency, high throughput IPC between functional components. As a result, scalability and additional functionality is limited to costly and proprietary "in the box" solutions. For example, web-switches predominately use proprietary internal interconnects and inter-process communication (IPC) to add incremental functionality.

4. Web switch appliances tend to be proprietary closed systems that do not allow the development of new services by a large community of developers.

5. Rapidly increasing network speeds and usage demands create a greater need for an efficient and general-purpose method to independently scale packet switching capacity, packet processing, and network services.

Software overhead in server operating system network stacks, specifically sockets and TCP/IP, has been identified and studied extensively over the last ten years. Efficiencies have been added by tuning the server resident software and protocol stacks and by incremental offloading of computation intensive portions of the network stack into specialized hardware on the Network Interface Card (NIC) [25, 27, 33]. These optimizations, along with increases in server processing speeds, have improved network overheads but at too slow of a rate compared to increasing network throughput and Internet application demands [15].

To overcome these inefficiencies, we investigated the use of a Virtual Interface (VI) Architecture enabled SAN fabric as the network interconnect between the web-switch and the front-end servers. This investigation was prompted partially by the results of our previous VI Architecture related research showing that use of VI Architecture provided significant performance and efficiency benefits over standard server networking interfaces, even when used with higher level socket interfaces [16]. As an experiment, we ran a simulated web transaction performance test in order to re-evaluate TCP/IP and VI Architecture performance characteristics on modern software and hardware. The test comprised of iteratively sending a 256-byte message, simulating an HTTP *get* request from a client, followed by receiving a reply message of varying sizes from the server. The transaction latency and the server host CPU cycles spent per transaction were measured.

In the experiment, we used Giganet cLAN product [31] as the native VI Architecture (nVIA) and M-VIA [11] over Gigabit Ethernet as the emulated VI Architecture (eVIA). TCP/IP related tests were run over Gigabit Ethernet. The results of this experiment are shown in Figure 2. As the graph illustrates, for the same reply size, nVIA and eVIA are able to achieve 2-2.25x and 1.3-1.6x better latencies than TCP/IP respectively. Furthermore, the CPU cycles spent per transaction for nVIA and eVIA were significantly lower than for TCP/IP (for large messages, an order of magnitude).

Another important characteristic of VI Architecture enabled SAN fabrics, is that they provide the mechanisms needed to enable efficient low latency IPC between distributed components in a system. By using these mechanisms, which consist of the standard VI Architecture interface (VIPL) and underlying reliable SAN wire transport, a lightweight protocol can be constructed to efficiently exchange control information and data between components.

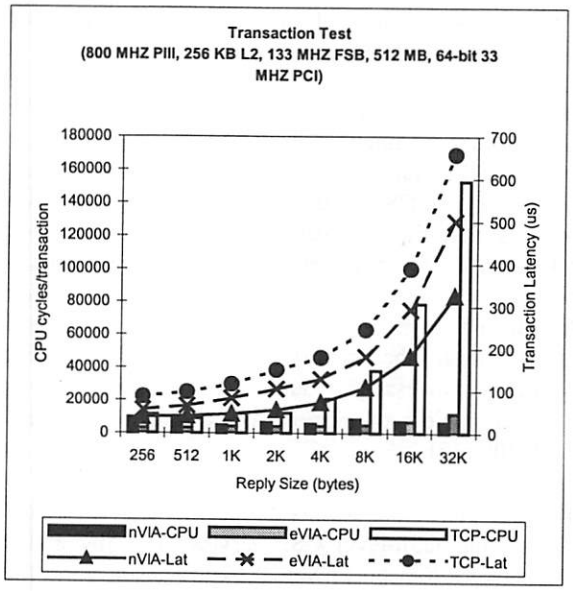


Figure 2: Transaction Test Results

Our idea was to research the use of VI Architecture enabled SAN fabric as the framework to create a distributed system architecture, termed CSP system architecture that would enable systems to be constructed entirely out of commodity building block components. The goals of this research being to address the issues raised earlier on the limited scalability, redundant packet processing, restricted functionality, and proprietary nature of existing web-switching systems deployed in today's data center.

3. System Architecture Overview

The CSP system architecture consists of multiple functional elements, or nodes, interconnected with a VI Architecture enabled SAN fabric. These elements can be enumerated to enable the construction of CSP systems with varying levels of functionality, scaling and performance. The decomposition of the CSP platform into a functional pipeline of building blocks allows scaling of each pipeline stage independently. Figure 3 illustrates the CSP system architecture. The functional elements of the CSP system architecture are described as follows:

System Area Network (SAN) and the Virtual Interface Architecture (VI Architecture): the SAN in the CSP system architecture provides the interconnecting fabric for all other elements in the system. CSP nodes attach to the SAN using the VI Architecture specification [20] compliant interface and the VIPL API [21]. A lightweight IPC mechanism, termed the "CSP Transport" is used to communicate control information and exchange data between the elements.

Network Node (NN): provides the LAN and/or WAN interface function in the CSP architecture. It performs the first level of processing in the functional pipeline of the CSP system by processing LAN/WAN packets and then forwarding them to other SAN nodes for further processing. Examples of NN packet processing include line-rate layer-3 packet forwarding, layer-4 load balancing, and TCP flow classification. The network node is optimized for fast packet processing and constructed using programmable network processors, such as the Intel® IXP1200 [32]

Proxy Node (PN): acts as the proxy between remote clients and high-level CSP services residing above the network transport layer. Proxy nodes perform the next level of processing in the CSP pipeline by terminating all client TCP sessions and subsequently communicating the associated data streams over the CSP transport between proxy and application nodes. Proxy nodes essentially de-couple network transport protocol processing cycles from application node compute cycles in order to enable independent scaling of packet and application processing. The architecture of proxy nodes in the CSP system enables them to perform various higher-level functions on the application data. Examples of these functions might include HTTP proxy services, web content transformation, such as SSL, and support for content-based distribution of web services.

***Application Node** (AN)*: hosts well-known applications, such as a web, mail or directory services. The application node is built using standard high-density server hardware and runs standard operating systems and applications. The application node utilizes the CSP transport to bypass the kernel-resident network stack during communication with proxy nodes (and/or possibly other SAN-enabled nodes).

Management: The CSP architecture defines a management function that provides dynamic resource discovery and configuration services, along with network policy management. As a central location for resource and policy information, the management function is assumed to be configurable in a redundant manner. An example of the CSP management function would be a pair of nodes in a CSP system that acts as a repository and distribution center for system configuration and packet processing policy.

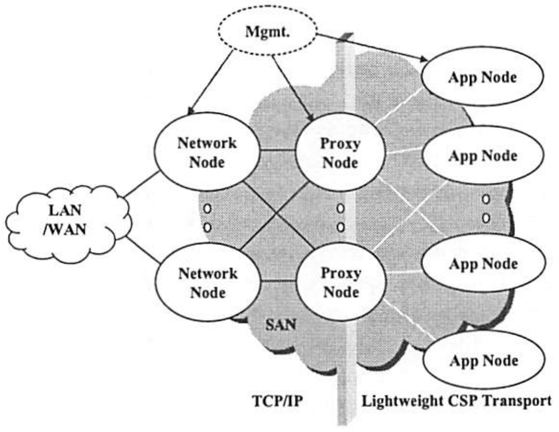


Figure 3: CSP System Architecture

4. CSP Solutions

The main focus of CSP is to provide solutions to the network edge related problems described in Section 2. This section describes main techniques and solutions developed for the CSP architecture.

4.1 SAN Tunneling

In a distributed functional pipeline, one function of the network node is to forward packets between the external network (LAN or WAN) and the SAN. In the CSP architecture, packets are communicated between a network node and a proxy node through a SAN tunnel (VI) based IPC mechanism. This is accomplished by encapsulating network packets with SAN packet header information. This feature is important in order to allow the proxy node to process TCP/IP packets at user level using VIs. Furthermore, SAN tunneling enables communication of additional information in a VI packet header that can be used in TCP/IP header compaction, TCP flow identification, and TCP checksum off-loading.

4.2 Flow Labeling

In order to eliminate redundant packet header processing and to enable efficient lookup for TCP packet flows, we developed a VI Architecture based TCP flow labeling technique. This technique is comprised of tunneling encapsulated TCP packets over VI connections, a simple TCP flow-identifier-based lookup scheme, and TCP/IP header compaction. The use of TCP flow labels enables simple indexing based lookup (constant time lookup) schemes on the network, proxy, and application nodes. On the proxy node, flow labels eliminate the need to maintain a separate TCP transmission control block (TCB) cache for lookup, enabling it to outperform other TCP TCB lookup schemes used in practice. On the network node, TCP flow labels are used on outbound TCP packet flows to simplify packet forwarding information lookup.

In contrast to other flow labeling schemes in use by the web switches, this scheme carries the TCP flow identification information all the way to the end stations (proxy nodes and application nodes) and thus reduces redundant TCP/IP header processing. TCP/IP header compaction is used to allow space for additional VI packet header bytes in the encapsulated TCP packet without having to fragment the packet or reduce the typical TCP maximum segment size (MSS) for like media, i.e. Ethernet.

During the initial TCP connection establishment, a flow identifier is generated by the proxy node and communicated to the network node as a part of an outgoing TCP SYN/ACK or TCP SYN packet. During the connection tear down, the proxy node informs the network node about the termination of the flow in the outgoing TCP FIN/ACK or last TCP ACK packet. For the incoming traffic with known TCP flows, the network node performs the lookup and TCP/IP header compaction (optionally), sets the flow identifier field, and tunnels the rest of the packet to the appropriate proxy node. Upon receiving a compacted packet for a known TCP flow from the proxy node, the network node performs the flow identifier based lookup, constructs IP header, and transmits it on an appropriate LAN port. The proxy node maintains a table of pointers to TCBs. It uses the flow identifier as an index to the table and this allows constant time TCB lookup for an incoming TCP packet with the flow identifier.

4.3 Distributed Network Services

The CSP architecture enables two levels of network services. At the first level, network nodes can perform layer-2 to layer-4 (L2-L4) processing and at the next level proxy nodes can perform layer-5 to layer-7 (L5-L7) processing. Compared to existing 'in-the-box' solutions offered today [26, 28, 29, 30, 34, 35], the advantages of CSP distributed network services are:

- They provide scalable, open, and cost-effective solutions as they are built using VI Architecture interfaces, programmable network processors, and standard rack-mounted servers.
- Distribution of service functionality allows the components to be independently scaled and optimized.

An example of this type of services is distributed network load balancing, where the network nodes perform L4 load balancing across the proxy nodes either using Network Address Translation (NAT) or IP tunneling and the proxy nodes perform application level load balancing (L5-L7) across the application nodes. Thus, the network nodes along with the proxy nodes can be viewed as a distributed network switch performing L4-L7 load balancing.

4.4 SAN Proxy Service

We developed a new CSP core service on the proxy nodes called *SAN Proxy* to mitigate the OS based TCP/IP processing bottleneck on the application nodes. The SAN Proxy is a transport layer proxy service developed for decoupling TCP/IP processing from application processing. The SAN Proxy independently manages TCP/IP connections with the network clients and SAN channels (VIs) with the application nodes, relays TCP/IP byte streams arriving from the network clients to the application nodes using a lightweight protocol, and communicates the lightweight protocol data arriving from the application nodes to the appropriate network clients using TCP/IP. In this service, TCP flows are terminated on the proxy nodes and only the data above the TCP layer is communicated to/from application nodes over reliable VI connections. As the network nodes and proxy nodes communicate TCP/IP packets over SAN tunnels, the proxy nodes can process TCP/IP packets at the user-level without any OS intervention.

This isolation and decoupling of TCP/IP processing from application processing provides the following advantages:

- The control traffic generated due to TCP/IP processing no longer interferes with the applications because it does not propagate beyond the proxy nodes. The protocol processing overheads on the application nodes can be significantly reduced because of the use of lightweight protocol (CSP transport) to communicate data.

- Since the proxy nodes are not constrained by the legacy API (sockets), OS environment, and the hardware platform, they can be optimized to meet both TCP/IP and SAN protocol processing demands and can be scaled independently from application nodes.

- One or more application nodes can share the proxy nodes. Thus, proxy nodes can be used to perform L5-L7 policy execution across the application nodes in addition to off-loading protocol processing.

- Because the SAN Proxy handles all packets moving through the system, it is the logical place to add higher level services such as caching, firewall, and content transformation.

Figure 4 shows the flow of packets for a HTTP/1.0 transaction on the CSP with TCP/IP termination and protocol translation at the proxy node, illustrating the simplified control traffic at the application node.

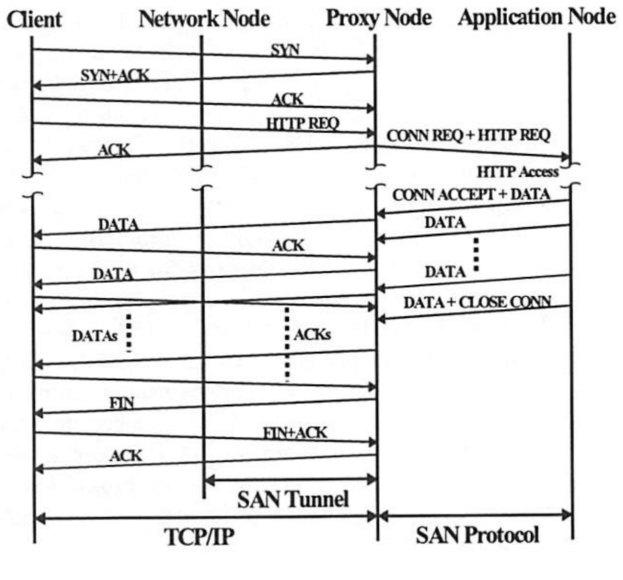


Figure 4: Example of a HTTP/1.0 Transaction with TCP/IP Termination on the Proxy Node

4.5 Application Support

Application support focuses on developing an efficient communication interface for front-end Internet applications. Internet applications commonly use a socket-based programming paradigm for communications. Our solution involves work in two major directions: i) support for existing legacy applications that use the BSD sockets API; ii) understand and overcome the communication constraints imposed by legacy API. DASockets [5] and Windows Sockets Direct Path (WSDP) [23] are efforts with similar goals in mind.

5 CSP System Evaluation

A combination of system prototyping and simulation was being used for evaluation of the CSP performance, scalability, and for architectural validation. The combination provided the proof-of-concept as well as proof-of-architectural implementation for CSP. In this methodology, prototyping with off-the-shelf hardware and software components was focused on dealing with the real world problems of implementing a CSP system with existing technology and on providing details on SAN protocols, message formats, and timing information. The prototype results were used in conjunction with the simulation models to evaluate overall system performance, identify bottlenecks, evaluate messaging flows, test flow control mechanisms, and validate improvements.

5.1 Prototype Implementation

This subsection describes a prototype implementation of the CSP. The prototype implementation provides practical experience in building a CSP system and provides a functional demonstration vehicle. The following subsections describe in detail each component of our CSP prototype.

5.1.1 SAN and CSP Transport

In order to construct the prototype using existing technologies, we used Gigabit Ethernet as the switched interconnect between the elements of the system. To provide a VI Architecture compliant interface within each element of the system, we used the Modular-VI Architecture (M-VIA) [11] emulation software developed at Lawrence Berkeley National Labs. The M-VIA software emulates the VIPL 1.0 API semantics in kernel software in the absence of native VI Architecture hardware support. This allows us to develop the services and interfaces independent of the underlying SAN fabric, as long as it supports the VIPL interface.

It is noted here that for our prototyping and analysis purposes, we assume that switched Ethernet provides the reliable link semantics normally supplied by true SAN implementations. Also note that the emulated VI Architecture over Ethernet does not provide the same *level of performance as a true SAN with native VI Architecture hardware support.*

Our prototype CSP transport is a lightweight protocol optimized for reliable SAN interconnects. The CSP transport was derived from our earlier stream sockets prototype over VI Architecture [16]. It was modified to allow proxy nodes to access internal VI descriptors and register/deregister buffers. Furthermore, the ability to wait or poll on particular values of the immediate data field of a VI descriptor was added for supporting multiplexing of sockets over a VI on the application nodes. The CSP transport manages VI resources and buffers used for communication, and performs credit-based flow control. The use of a single VI connection between an application node and a SAN proxy can have serialization and single-point of failure problems. On the other hand, use of a VI per socket limits scalability. In the CSP prototype, application level communication endpoints (sockets) were multiplexed over a pool of VI connections. A cache of registered buffers was maintained by the CSP transport in order to reduce the cost of memory registration and de-registration.

5.1.2 Network Node

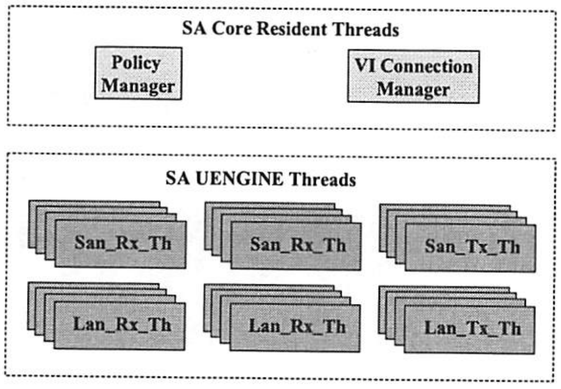


Figure 5: An IXP1200 based Network Node

The prototype CSP network node used an Intel® IXP1200 network processor [32]. The IXP1200 is a single chip network processor with interfaces to external memories and media access devices. The internal architecture of the IXP1200 consists of the following functional units; six multi-threaded micro-coded RISC engines (UENGINES), one StrongARM® core processor (SA_CORE), SRAM and SDRAM memory interface units and an external bus interface unit (IXBUS) which is used to connect media access controllers. The prototype IXP1200 configuration used in the CSP prototype consisted of a single IXP1200 chip, 32 Megabytes of external SDRAM, 2 Megabytes of external SRAM, one eight port (quad) 10/100 Mbps Ethernet MAC for attaching to the LAN, and one two port Gigabit Ethernet MAC for attaching to the SAN.

The network node software architecture consists of twenty-four UENGINE threads for packet processing and two SA_CORE resident management threads. The SAN interfaces were made VI Architecture compliant by porting and running the M-VIA stack on the UENGINE threads. The UENGINE threads perform one of four packet processing operations; reception and processing of packets received on the LAN interface (Lan_Rx_Th), reception and processing of packets received on the SAN interface (San_Rx_Th), transmission of packets onto the LAN interface (Lan_Tx_Th), and transmission of packets onto the SAN interface (San_Tx_Th). Each of these operations is multi-threaded and the threads are distributed onto multiple UENGINES in order to provide wire speed packet forwarding between LAN and SAN ports. Figure 5 shows the IXP1200 based network node.

5.1.3 Proxy Node

In our prototype, we used standard rack mounted 800 MHz Pentium® III processor based servers running Linux OS (version 2.2.14-20) as the proxy nodes. The Proxy node uses user-level TCP/IP (derived from [10]) to communicate with LAN/WAN clients. On the behalf of clients, it interacts with the application nodes using the CSP transport. In our prototype, we only implemented the SAN Proxy service and a simple L4 load-balancing scheme on top of it.

In our implementation, the SAN Proxy service is a user-level multithreaded application that uses a pool of worker threads. Each worker thread is specialized for performing a subset of functions associated with TCP/IP processing, protocol translation/decoupling, SAN protocol processing, initial discovery mechanism, and L4-L7 policy execution. The SAN Proxy maintains two separate pools of registered buffers (one for incoming traffic and another for outgoing traffic). By maintaining a memory buffer structure that tracks the state of each registered buffer, the SAN Proxy achieves zero-copy translation. Figure 6 shows an architectural view of a proxy node along with a network node and an application node running a legacy application.

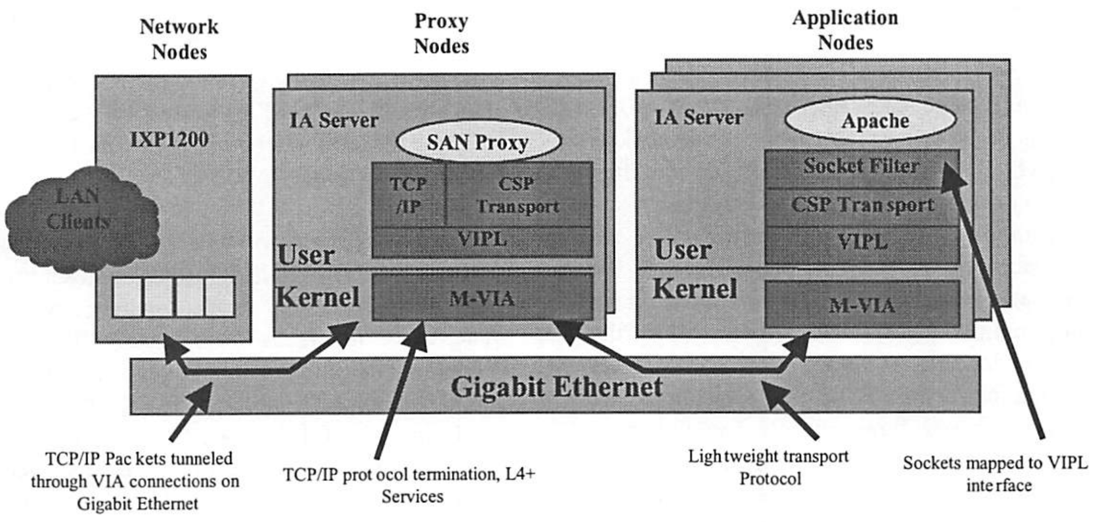


Figure 6: TCP/IP Termination Using Proxy Node

5.1.4 Application Node

Our initial approach was to enable existing applications without modifying the OS or applications. In our prototype, we introduced a *Socket Filter* module to intercept all socket-related function calls made by the legacy applications, and map them into appropriate CSP transport messages. In an environment such as Windows™ NT, the socket filter module could be loaded dynamically as a dynamically loadable library (DLL). However, due to the lack of DLL support in Linux, the application has to be recompiled and linked to the socket filter library.

We assumed that a majority of Internet server applications follow a predictable logic flow for socket-related calls, and we can consistently map them to the CSP transport messages. To simplify prototyping, we worked with those versions of applications that are multi-threaded, since VI resources cannot be shared across multiple processes. To date, we have successfully used the socket filter to enable an ftp server (betaftpd-0.0.7) and two web server applications (thttpd –2.16 and Apache-2.0a-dev3).

In order for a legacy application to preserve its host OS descriptors on the application nodes, the socket filter partitions 16-bit file descriptor (*fd*) space into system *fds* and transport *fds*. Furthermore, a *fd* for listening on a well-known port is obtained from host OS. The socket filter uses flow identifiers for both actively and passively opened sockets. For passively opened sockets, flow identifiers supplied by the proxy nodes are used as new socket *fds*. For an actively opened socket, a mapping between a socket *fd* and the corresponding flow identifier is used as the socket *fd* needs to be generated prior to the connection establishment.

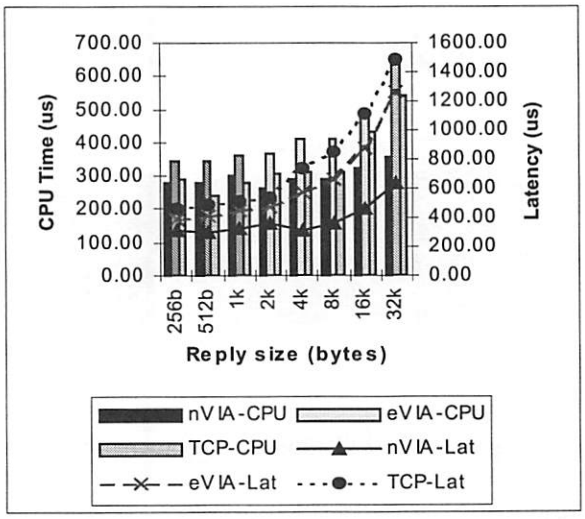


Figure 7: Performance of Apache over Various Transports

To ensure that real applications can truly benefit from the use of lightweight CSP transport, we determined the performance of the application node in isolation. Figure 7 compares the performance of an application node responding to HTTP/1.0 GET requests of various file sizes using Apache over TCP/IP, eVIA, and nVIA. A simple client was used to drive sequential, single-stream HTTP GET requests to Apache over TCP. The same requests were issued to Apache over eVIA and Apache over nVIA hardware by using a simulated proxy (and client) over the CSP transport. The simulated proxy replicates the behavior of the proxy node and issues the necessary CSP transport messages. To ensure a fair comparison, the maximum packet size was set to 1514 bytes (including headers) and non-blocking writes were used in all cases. It should be noted that larger Maximum Transfer Sizes (MTS) up to 64K (as used in our experiments related to Figure 2) are more commonly used in SANs, thereby allowing for better performance. Furthermore, our prototype has to work under the restrictions imposed by legacy applications with substantial overheads added by the socket filter module. For the file sizes we considered, Apache over nVIA achieved 1.5-2.9x improvement in the latency. Furthermore, nVIA incurred less CPU overhead than TCP/IP.

5.1.5 Management Node

The main functions of the management node in our prototype is to maintain a centralized CSP information database, set policies on other nodes, and facilitate initial discovery operations. Other nodes register with the management node and inform it of their capabilities. As a part of initial discovery, the management node provides the proxy nodes and application nodes information about the network nodes and proxy nodes respectively. L2-L4 policies on the network nodes and L4-L7 policies on the proxy nodes are set after initial registration phase.

5.2 CSP Simulation Model

Development of the CSP simulation model and initial CSP system simulation studies were performed in parallel with the development of the actual CSP prototype. This allowed us to do early "pre-prototype hardware" studies of SAN fabric and CSP system configurations. The SAN fabric simulations were focused mainly on developing the SAN fabric infrastructure that would support the interconnect requirements of the CSP architecture, would scale to clusters of at least 64 nodes, and would allow configuring the SAN for either Ethernet or Infiniband™ Architecture operation. This enabled testing various CSP configurations with various SAN protocol and fabric topology options. The CSP system simulations were focused on evaluation of the CSP web services performance on various system configurations and workload scenarios. Since we didn't have all the critical CSP prototype software tuned for performance and test results at the time this paper was written, we used a three-step process to characterize as much of a range of performance, scaling, and configuration information as we could.

The test (CSP) configuration used for our system simulations consisted of; two network nodes, each with a Gigabit Ethernet link to the external client network, two proxy nodes with two SAN ports each, four application nodes with one SAN port each, and a 10 port SAN switch interconnecting all the nodes. The SAN links were modeled as 100 meter, full-duplex links, at 1.25 Gbps for Ethernet and 2.5Gbps for Infiniband™ Architecture. SAN messages were segmented into 1514 byte frames for Ethernet and 282 byte frames for Infiniband™ Architecture.

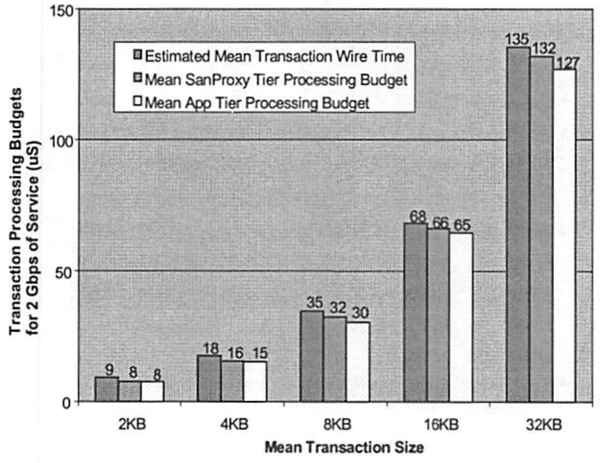


Figure 8 CSP Processing Time Budgets

For the first step in characterization, system simulation was used to empirically determine the processing budgets for the proxy and application tiers of the test configuration. These budgets were compared with the calculated mean wire time for each transaction size on the two output links from the network nodes to the clients (see Figure 8). The budgets were derived using constant streams of HTTP/1.0 GET requests for power-of-two transaction size from 2KB to 32KB. Each budget shows the mean transaction processing time the test configuration must achieve in the proxy or application tier to fully utilize the bandwidth of the two Gigabit Ethernet links to the client network, given a constant stream of the corresponding transaction size.

The second step was to establish the 800 MHz Pentium® III Processor based server we use in our prototype system as a basic unit of measure of processing capacity (*P*) for characterizing the application tier (we have not yet acquired the data we need to do this for the proxy tier). In Figure 9, we graphed the application tier budgets from Figure 8 and the mean transaction processing times for the native VI Architecture case in Figure 7. Then we divided the native VI Architecture based processing times by the application tier processing budgets to create the curve showing the equivalent number of *P* required in the application tier to achieve the maximum throughput for each mean transaction size. It shows that it would take the equivalent of 33*P* to sustain the full 2 Gbps throughput with a stream of 2KB web transactions or 2.8*P* with a stream of 32KB transactions.

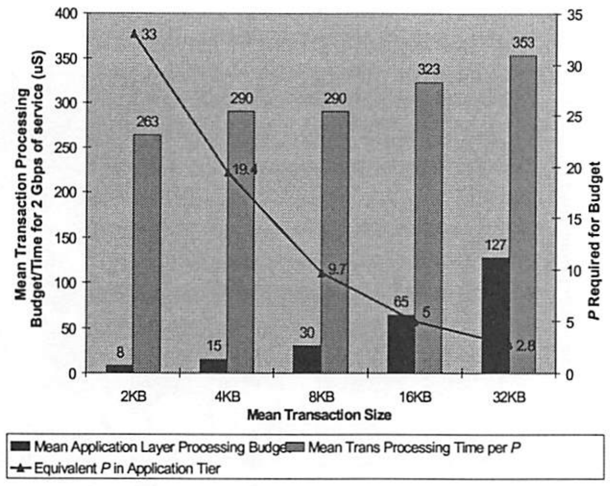


Figure 9: Equivalent *P* in Application Tier for 2 Gbps of Service

The third step was to input the appropriate processing times into our test model to simulate CSP configurations with the equivalent processing capacity of each of the points graphed on the curve in Figure 9. For each of these equivalent-processing capacities, we applied a web transaction workload consisting of transaction sizes randomly selected from the SPECweb99 static distribution (a mean transaction size of 14,384 bytes). Requests were generated by the clients in an exponential distribution with a mean rate of 20000 requests per second. Each test was run until 50,000 transactions were completed. The results of these simulations were used to produce the chart shown in Figure 10. The bars graph the total system throughput in transactions per second (TPS) and the curve graphs the TPS per *P* equivalent processing capacity.

The chart in Figure 10 shows that the performance of our test configuration peaks at the equivalent processing capacity of about 10*P* for 2 Gbps of service. It also shows that, although the system throughput drops significantly when the processing capacity is reduced to the equivalent of ~3*P*, the TPS per *P* is still increasing. If we consider only the cost/performance of the application tier, this suggests the best cost/performance CSP configurations for this workload will have the equivalent of 1*P* to 2*P* per gigabit of service. However, if such things as the cost of the links to the clients network are considered, one may need anywhere from 3*P* to 5*P* per gigabit of service to maximize the cost effectiveness of each network link. Additional processing loads such as dynamic objects, SSL processing, or content transformations will also drive up the processing capacity requirements.

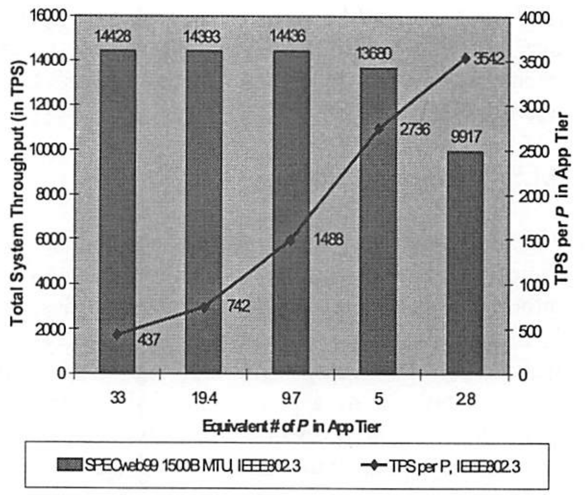


Figure 10: SpecWeb99 Static Distribution Performance

By including complete benchmark processing loads in the performance simulations and by collecting the critical performance data for each prototype node, we will be able to fully characterize the CSP in the future and enable easily optimizing configurations for various applications and cost/performance tradeoffs.

6 Conclusions and Future Work

In this paper, we described the CSP system architecture for scalable Internet and communication services. The distinguishing features of the CSP architecture being; SAN as the system interconnect and VI Architecture as a low-overhead interface to SAN, use of commodity building block components, decomposition of network services into distributed functional pipelines, and the de-coupling of network protocol processing from end-point application processing. We described several CSP solutions, specifically the use of SAN tunneling and TCP flow labeling to reduce redundant packet processing, the CSP transport to provide an efficient "out of the box" IPC mechanism, the SAN proxy service that terminates client TCP/IP flows and maps them to the CSP transport, and the distribution of network services to allow scalability by service function.

Our prototype implementation and system simulation results have offered us significant insights into the viability, performance and scalability of web service systems based on the CSP architecture. The current prototype, which is functionally complete and operational, but not yet tuned for optimal performance, has allowed us to understand the difficulties and limitations imposed when constructing CSP systems out of available commodity building block components. The CSP simulation models have allowed us to extend past these limitations in order to gain the knowledge needed to architect future interconnect and component technologies better suited for CSP like systems. The following is a summary of the insights gained and issues raised during both the CSP prototype and modeling efforts:

- It is possible to construct a CSP type system using commodity-based components that can be scaled by function and processing capability to achieve a desired system throughput and cost point. To date, our CSP simulation studies have shown significant evidence that the CSP systems can be built from commodity building blocks and provide the necessary characteristics and scalability to be competitive with more specialized solutions. Unfortunately, at this time we have not acquired all of the critical prototype data needed to complete our analysis and fully characterize and contrast our solution against others. We plan to do so in future.
- Prototype benchmarking and simulation results showed that the use of a light weight CSP transport had performance benefits over traditional network transports, especially on components running traditional operating system stacks, such as application nodes. These benefits are largely due to the underlying VI Architecture and SAN interconnect technologies. The challenge lies in how to balance these benefits with the constraints imposed. For example, our early investigations showed that significant performance benefits could be achieved by using VI Architecture interfaces for web type transactions. Later prototyping showed that utilizing the VI Architecture interface efficiencies while still maintaining application transparency and compatibility of existing Internet application interfaces, i.e. BSD sockets, was very challenging. In particular, the *select()* paradigm, used by socket-based applications, did not map efficiently to the queued real time signals available in message passing primitives provided by VI Architecture interfaces.
- The current CSP prototype and system model use a proxy node architecture that required all TCP control and data traffic be stored and forwarded through it. This presents challenges in how to balance memory capacity, I/O throughput, and processing capacity with price/performance and functionality on the proxy node.
- CSP System simulations have shown very aggressive processing budgets are needed for both the proxy and application node components to achieve maximum system capacity. Current generation components are not optimized for CSP traffic patterns or processing budgets and therefore require higher degrees of CSP node replication or component cost. By developing a set of server silicon components that are optimized for the CSP architecture and developing key hardware functionality, we believe a much better cost/performance solution can be achieved.

Going forward, we plan to address the constraints imposed by the legacy sockets API as well as explore legacy-free application interfaces that can take full advantage of the CSP transport optimizations (such as user level I/O, asynchronous I/O, and zero copy semantics). Ultimately, we hope to develop a set of OS independent high-performance APIs to facilitate this transaction. To address the proxy node store and forward issue, we plan to investigate alternative control and data flow methods, such as "third party data transfers".

Other areas for future work are extending the CSP architecture to support SAN-aware file systems as defined in the direct access files system (DAFS) specification [6] and also extending the CSP architecture to interface into more traditional SAN-aware back-end applications, such as distributed databases.

To date we have focused heavily on basic web traffic. In the future, we would like to explore more communication intensive applications focused on a mixture of voice, rich media and data with additional infrastructure services such as SSL and Firewalls. We believe the CSP architecture is a relevant step in the evolution of the converged data center networks of the future.

References

1. Apostolopoulos et al. "Design, Implementation and Performance of a Content-Based Switch", *In Proc. of IEEE INFOCOM 2000,* 2000.
2. Camarda et al. "Performance Evaluation of TCP/IP Protocol Implementations in End Systems", IEE Proc. Comput. Digit. Tech. Vol. 146, Jan 1999.
3. A. Cohen et al. "On the Performance of TCP Splicing for URL-Aware Redirection", *Proc. of the 2nd USENIX Symp. on Internet Technologies & Systems,* 1999.
4. Edith Cohen et al. "Managing TCP Connections under Persistent HTTP", *Proc. of the Eighth International World Wide Web Conf.,* 1999.
5. DASockets Protocol Specification, Version 0.6, Network Appliance, Inc., 2000.
6. Direct Access File System (DAFS). http://www.dafscollaborative.org/.
7. D. Dunning, G. Regnier, G. McAlpine et al. "The Virtual Interface Architecture", *IEEE Micro*, Vol. 3, No. 2, pp. 66-76, 1998.
8. A. Fox et al. "Cluster-Based Scalable Network Services", *Proc. of the sixteenth ACM symp. on Operating systems principles,* pp. 78-91, 1997.
9. Infiniband™ Arch. Spec. Vol 1 & 2. Rel. 1.0. www.infinibandta.org/download_spec10.html.
10. KA9Q NOS TCP/IP. http://people.qualcomm.com/karn/code/ka9qnos.
11. M-VIA : A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via.
12. D. Maltz and P. Bhagwat. TCP Splicing for application layer proxy performance. Technical Report RC 21139, IBM, March 1998.
13. D. Maltz and P. Bhagwat. "Improving HTTP Caching Proxy Performance with TCP Tap", *In Proc. of HIPPARCH'98*, pp. 98-103, 1998.
14. V. Pai et al. "Locality Aware Request Distribution in Cluster-based Network Servers", In *Proc. Architectural Support for Programming Languages and Operating Systems*, 1998.
15. Greg Regnier, Dave Minturn, et al., Internet Protocol Acceleration Techniques, Intel Developer Forum, February 1999.
16. H. Shah, C. Pu, and R. Madukkarumukumana. "High Performance Sockets and RPC over Virtual Interface (VI) Architecture", *In Proc. Third Intl. Workshop on Communication, Architecture, and Applications for Network Based Parallel Computing*, pp. 91-107, 1999.
17. Evan Speight, Hazim Abdel-Shafi, and John K. Bennett. "Realizing the Performance Potential of the Virtual Interface Architecture", *In Proc. of the 13th ACM-SIGARCH International Conference on Supercomputing,* June 1999.
18. Junehwa Song et al. "Design Alternatives For Scalable Web Server Accelerators", *In Proc. of the IEEE International Symp. on Performance Analysis of Systems and Software*, April 2000.
19. Oliver Spatscheck et al. "Optimizing TCP Forwarder Performance", In IEEE/ACM Tran. of Networking, Vol. 8, No. 2, April 2000.
20. Virtual Interface Architecture Specification, Version 1.0, http://www.viarch.org/.
21. Virtual Interface Architecture Developer Guide, Intel Corporation, http://developer.intel.com/design/servers/vi/.
22. G. Welling, M. Ott, and S. Mathur. "CLARA: A Cluster-Based Active Router Architecture", *Hot Interconnects 8,* pp. 53-60, 2000.
23. Windows Sockets Direct Path for System Area Networks. Microsoft Corporation, 2000.
24. C. Yang and M. Luo, "Efficient Support for Content-Based Routing in Web Server Clusters", *Proc. of the 2nd USENIX Symposium on Internet Technologies & Systems,* 1999.
25. Alacritech, Alacritech Server Network Adapters. http://www.alacritech.com/html/products.html.
26. Alteon WebSystems, Alteon Web Switching Products. http://www.alteonwebsystems.com/products/.
27. Alteon WebSystems, Next Generation Adapter Design and Optimization for Gigabit Ethernet. http://www.alteonwebsystems.com/products/whitepapers/adapter
28. ArrowPoint Communications, ArrowPoint Content Smart Web Switches. http://www.arrowpoint.com/products/index.html.
29. Cisco Systems, Cisco Local Director. http://www.cisco.com/public/product_root.shtml.
30. F5 Networks, BIG-IP Products. http://www.f5labs.com/f5products/bigip.
31. Giganet, Inc., Giagnet cLAN Product Family. http://www.giganet.com/products/.
32. Intel® IXP1200 Network Processor. http://developer.intel.com/design/network/products/npfamily/ixp1200.htm.
33. Interprophet Corporation. http://www.interprophet.com/.
34. NetScaler, WebScaler Internet Accelerator. http://www.netscaler.com/products.html.
35. Resonate, "Central Dispatch 3.0 – White Paper", http://www.resonate.com/.

The Age Penalty and its Effect on Cache Performance

Edith Cohen
AT&T Labs–Research
180 Park Avenue, Florham Park
NJ 07932, USA
edith@research.att.com

Haim Kaplan
School of Computer Science
Tel-Aviv University
Tel-Aviv 69978, Israel
haimk@math.tau.ac.il

Abstract

Web content caching is recognized as an effective mechanism to decrease server load, network traffic, and user-perceived latency. An HTTP compliant cache associates with each cached object an expiration time calculated according to directives set by the object's origin server. The cache incurs a *miss* when it has no cached copy of a requested object or when the existing copy had expired (is not fresh). Upon a miss, the cache needs to fetch or validate a copy through exchanges with another cache with a fresh copy or the origin server. Thus, misses generate traffic and prolong service times.

Caches are deployed as proxies, reverse proxies, and hierarchically and as a result, caches often serve other caches. As this happens, content *age* at higher-level caches, in addition to availability and freshness, emerges as a performance factor. The *age* of a cached copy of an object is the elapsed time since fetched from the respective origin. Fresh cached copies of the same object can have different ages and older copies typically expire sooner. Therefore, a proxy cache would suffer a higher miss rate if it receives older objects (e.g., from a reverse-proxy cache). Similarly, reverse-proxy caches that serve proxy-caches receive more requests than an origin server would have received. We refer to the increase in miss rate due to age as the *age penalty*. We use trace-based simulations to measure the extent of the age penalty for content served by content delivery networks and large caches. Even though the age penalty had not been considered previously, we demonstrate that it can be significant, and moreover, can highly vary under different practices.

1 Introduction

Caching and replication of Web contents are widely deployed mechanisms for reducing Web servers load, network load, and user-perceived latency. The cache effectiveness depends on the lifetime durations of cached copies. Web caching is governed by HTTP, which associates with each cached copy of an object a freshness expiration time after which it is considered stale. When a request arrives and there is no cached copy (*content miss*), the cache attempts to obtain a fresh copy through another cache or the origin server. When a request arrives and there is a stale cached copy of the object, the cache must attempt to validate it (ask for validation or a modified copy) before serving it to the user Validations are performed through a conditional (`If-Modified-Since` or `E-tag` based) `GET` request and involve communication with the origin server or another cache. A large fraction of validation requests return "unmodified" (we term such requests *freshness misses*), but very often the latency they incur is comparable to that of a full-fledged content miss. Thus, both content and freshness misses decrease the cache effectiveness and are important to avoid.

Caches determine an expiration time for a cached copy by computing its *freshness lifetime* and its (approximate) *age*. A copy becomes *stale* when its age exceeds its freshness lifetime. The freshness lifetime is essentially determined by the origin server as it is computed using directives and values found in the object's response headers. If explicit directives are not provided, a heuristic is used. The copy's age is the elapsed time since it was sent by its origin server. When a copy is obtained from another cache it has a positive age and thus, typically a shorter time-to-live (TTL) than would have been if fetched directly from the origin server.

Caches are being deployed throughout the network. Proxy caches are placed in ISPs networks close to clients and reverse proxy caches are placed near network exit points and cache objects for hosted Web sites. Content delivery services (CDNs) such as Akamai and Sandpiper place caching servers in multiple locations [1, 10]. Therefore, it is becoming increasingly more common that (explicitly or transparently) content is served to a cache (such as a proxy cache) from another cache (e.g., a reverse proxy). As this happens, another performance factor emerges, namely, the *age* of copies at higher-level caches. If an older copy of an object is received by the low-level cache, the cached copy expires sooner and thus is less likely to remain fresh till a subsequent request arrives for the same object. Therefore, low-level caches that receive older copies incur more misses. We refer to the relative difference in the number of freshness misses incurred by a cache when it forwards requests to a non-authoritative (high-level cache) vs. authoritative (origin server) source as the *age penalty*.

Cache performance issues related to content age had been by and large overlooked. Our main contribution is introducing the age penalty issue and assessing its magnitude. Aging issues take a different spin for content served by CDNs. CDNs such as Akamai act as reverse proxies, but have tighter relation with their clients (Web sites). In particular, they deploy a consistency mechanism other than HTTP/1.1 between them and their clients. Downstream client caches, however, still use the HTTP response headers to determine the object lifetime. The interaction of two coherence mechanism gives rise to different practices that greatly affect object age, and as a result, the effectiveness of client caches, user-perceived latency, and traffic. We discuss observed CDN practices and measure the associated age penalty.

In Section 2 we overview HTTP freshness control and its usage, and introduce aging issues. Section 3 discusses our traces and experimental methodology. Section 4 contains simulation results measuring the age penalty for objects served from a high-level cache. Section 5 is concerned with the age penalty for objects served by content delivery networks. We conclude in Section 6.

2 Freshness control

Cache freshness control mechanisms are discussed in the specification of HTTP/1.1 and its predecessors. Caches should determine the freshness period conservatively, using parameters in the response headers of the object. If the header does not include specific directives, then it is suggested that caches apply a heuristic, by first matching the URL against some regular expression (e.g., according to suffix which usually indicates the content type), and then determining the freshness lifetime to be some fraction of the elapsed time between the object's date and its last modification time. We provide a quick overview of freshness control at caches. For further details see [5, 2, 11, 8, 7].

The following response headers are used by Squid [11] for freshness control:

- DATE. A time stamp indicating when an object is sent by the **origin server**. All origin servers that have clocks must provide a DATE header. An object received without a DATE header must be assigned one by the recipient if the object will be cached. The DATE header of an object is updated after a 304 (not modified) response to an `If-Modified-Since` GET request. This header is an end-to-end header not supposed to be changed by intermediate caches.
- PRAGMA: NO-CACHE (HTTP/1.0) or CACHE-CONTROL: NO-CACHE (HTTP/1.1).

Explicit directive that the object is not to be cached. Response without this directive is considered cachable.

- CACHE-CONTROL: MAX-AGE (HTTP/1.1). Explicit freshness lifetime value assignment by the origin server in seconds. This value is typically fixed for copies of the object obtained at different times.
- EXPIRES: (HTTP/1.0). A time stamp beyond which the object stops being fresh. If MAX-AGE is also present then MAX-AGE takes priority. In practice, EXPIRES is often either set to be the current time or time in the past (implying freshness lifetime of zero), a far time in the future (years), or is calculated dynamically as DATE + T (providing for a freshness lifetime of T) and essentially behaves like a MAX-AGE directive.
- LAST-MODIFIED (HTTP/1.0). The time when the object was last modified by the origin server.
- E-TAG (HTTP/1.1). An identifier for the received version of the object. The E-TAG is generated at the origin and can be used for validation instead of a LAST-MODIFIED time.
- AGE (HTTP/1.1). The cumulative time an object spent in caches when received by the current cache. This response header is added and modified by caches and is present if all caches along the response path are HTTP/1.1 compliant.

HTTP/1.1 specification (and Squid) consider every object as cachable unless an explicit no-cache directive is present.[1] The freshness calculation for a cachable object compares the *age* of the object with its *freshness lifetime*. If the age is smaller than the freshness lifetime the object is considered fresh and otherwise it is considered stale. We refer to the difference between the freshness lifetime and the age as the time-to-live (TTL).

Squid calculates the age of an object as the difference between the current time (according to its own clock) and the time specified by the DATE header. This method of age calculation is also described in HTTP/1.1 specification. If an AGE header is present, the age is taken to be the maximum of the above and what is implied by the AGE header.

[1]There are few exceptions to this rule such as responses to requests with AUTHORIZATION headers and responses with VARY headers that are not yet supported by Squid and are very rarely used.

Squid implements freshness lifetime calculation according to HTTP/1.1 specifications as follows. First, if a MAX-AGE directive is present, the value is used as the freshness lifetime, and the TTL is the freshness lifetime minus the age (or zero if negative). Otherwise, if EXPIRES header is present, the freshness lifetime is the difference between the times specified by the EXPIRES and DATE headers (zero if negative), and thus the TTL is the difference between the time specified by the EXPIRES header and the current time (or zero in case this difference is negative). Otherwise, no explicit freshness lifetime is provided by the origin server and a heuristic is used: The freshness lifetime is assigned to be a fraction (denoted by CONF_PERCENT, HTTP/1.1 mentions 10% as an example) of the time difference (denoted by LM_AGE) between the time specified by the DATE header and the time specified by the LAST-MODIFIED header, subject to a maximum allowed value (denoted by CONF_MAX and is usually a day, since HTTP/1.1 requires that the cache must attach a warning if heuristic expiration is used and the object's age exceeds a day). Origin servers are supposed to keep their clocks fairly close to real time, and the age calculation assume that this is indeed the case.

2.1 Usage of freshness control

To get an idea for the distribution of freshness control mechanisms used, we took one of the logs we used (6 day NLANR cache trace, see Section 3) and performed separate GET requests to URLs appearing in the log. We then weighted the freshness mechanism by the number of requests to the object recorded in the log. 3.4% of requests were to objects with MAX-AGE specified. 1.4% were with no MAX-AGE header and with EXPIRES specified in a relative way (or to a time equal or before the one specified by the DATE header), and 0.8% were with EXPIRES specified in an absolute way. The vast majority, 70%, did not have either MAX-AGE or EXPIRES specified but had a LAST-MODIFIED header which allowed for a heuristic calculation of freshness lifetime. Other requests either had neither of these 3 header fields, were explicit noncachables (3%), or corresponded to objects with response headers other than 200 (when separately GETted), the most common of which was 302 (HTTP redirect).

Figure 1 plots CDF (Cumulative Distribution Function) of freshness lifetime values weighted by respective number of client requests. The majority of freshness lifetime values are 24 hours (86400 seconds), which is mostly due to the heuristic calculation (using LAST-MODIFIED) with a CONF_MAX setting of 24 hours. About 25% of values are

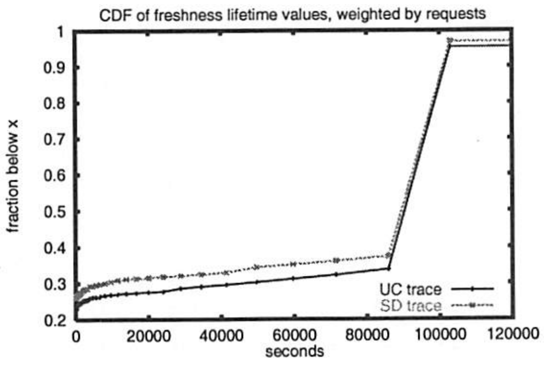


Figure 1: CDF of freshness lifetime values weighted by requests (cachable objects only)

0 (due to MAX-AGE or EXPIRES directive). The linearity of the line between very short freshness lifetime values and a value of 86400 (one day) is due to recently-modified objects and the heuristic expiration calculation (objects modified less than 10 days ago and a CONF_PERCENT value of 10%). These statistics also show that most freshness lifetime values are in effect *fixed*, being the same for copies subsequently obtained from the origin. The freshness lifetime value is fixed when it is computed using MAX-AGE, relative use of EXPIRES, EXPIRES value before the current DATE (freshness lifetime of 0), or with the heuristic when the object is sufficiently old to have freshness lifetime of CONF_MAX (that is, the elapsed time since its creation times CONF_PERCENT is at least CONF_MAX). The two cases without fixed freshness lifetime values are (i) recently-modified objects with no explicit freshness control directive, and (ii) the EXPIRES value is fixed (that is, the server returns the same value for later requests). As we shall see next, these distinctions are relevant for our study.

2.2 TTLs via different sources

We consider the TTL of a cached copy of an object at the time it is received by a cache. The age of a received copy, and thus, its TTL, may vary according to the source. A copy fetched through the origin is typically received with zero age whereas a copy obtained from a cache has positive age. [2] The relation between age and TTL depends on the applicable freshness control directive: With a "fixed" freshness lifetime value, (as with MAX-AGE header. dynamically-set EXPIRES header, or heuristic expiration for objects that were not modified very recently), the difference in TTL durations of two fresh copies is equal to the difference in their ages (the older copy becomes stale first). For objects with a "static" EXPIRES header, the TTL is the *same* for copies with different ages. For recently-modified objects with no explicit directives, the difference in TTL values of two copies is in fact *greater* than the difference in their ages.[3]

The following figures illustrate these gaps. Figure 2(A) plots the TTL for an object with MAX-AGE or "relative" EXPIRES freshness control when the object is served from its origin and when it is cached at a top-level cache. The illustrated scenario is such that the cache directly contacts an authoritative source only when its cached copy is stale. Figure 2(B) plots the same for objects with heuristic expiration based on LAST-MODIFIED header. If `no-cache` requests are received by the parent-

[2]Recall that the age of the copy is calculated as the maximum of the age and the difference between the current time and the DATE header value. Hence, if clocks at different hosts are synchronized, copies obtained from a cache should have positive age.

[3]Recall that when EXPIRES or MAX-AGE headers are not present, a heuristic is used to determine the freshness lifetime. The freshness lifetime is determined as a fraction of the difference between DATE and LAST-MODIFIED. In that case the TTL gets penalized twice for the copy's age: first, the freshness lifetime (fraction of the difference between DATE and LAST-MODIFIED subject to a limit) is smaller for older copies, and second, the age of the older copy is larger.

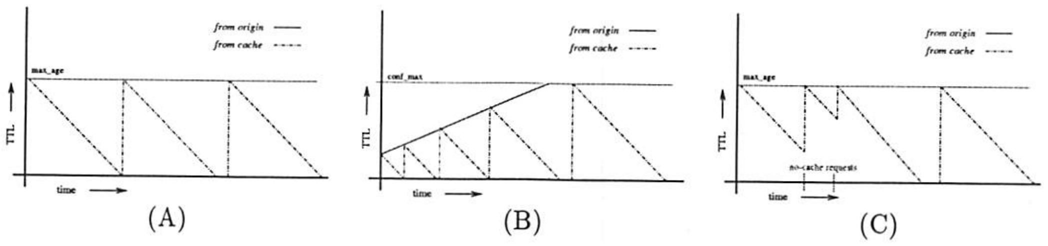


Figure 2: TTL for an object (i) when fetched from the origin and (ii) when fetched from a cache. (A) the object has MAX-AGE response header; (B) object has heuristic expiration based on LAST-MODIFIED response header, The slope of the line corresponds to CONF_PERCENT value and it levels off at CONF_MAX; (C) object with MAX-AGE response header when two client requests with a `no-cache` directive where received.

cache, the TTL would look as illustrated in Figure 2(C).

2.3 Age penalty

The age-penalty is measured by a what-if scenario (illustrated in Figure 3). We compare the performance of a (low-level) cache that forwards requests (transparently or explicitly) to high-level caches to the performance if all requests were forwarded to origin servers. We focus here on the freshness hit-rate at the cache. Generally, the age-penalty effect depends on the relation of inter-request times and freshness lifetime duration and kicks in when the low-level cache receives more than one request per freshness lifetime.

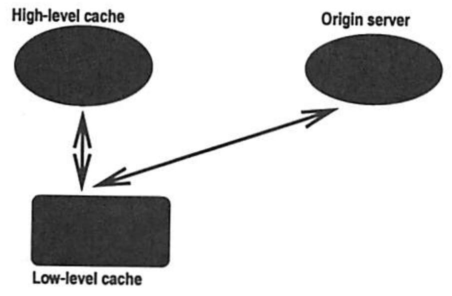


Figure 3: A low-level cache with choice of sources.

3 Experimental methodology

Our data included NLANR cache traces [6]. We used two 6 days traces from NLANR caches collected January 20th till January 25th, 2000 and between July 25 to July 29, 2000 (the second set of logs was used only in measuring CDN performance).

Our experiments required response header values in order to deduce freshness lifetime durations. As these are not typically logged by high-volume caches (including the NLANR traces), we separately performed GET requests shortly after downloading the trace. Timeliness and the distribution of freshness control directives and values (presented in Section 2.1) suggest that the projected values we obtained were fairly close to actual freshness lifetime values that would have been obtained at the time requests were logged.

We run simulation using the original trace and the projected values. The NLANR logs contain various labels that characterize each request by its type and cache performance. (Classification of hits and misses, presence of `no-cache` request header, whether the cached copy was stale or fresh, etc.). In the simulation we accounted for clients requests with `no-cache` header (forcing the cache to forward the request to the server even if the cache has a fresh copy of the object). We used a heuristic to determine on which requests content had changed (treating requests on which the content deemed fresh by the NLANR cache as "no change."). Our heuristic used the labels associated with the request and the logged size of the response to the client. In the simulation we applied the Squid object freshness model (HTTP/1.1 compliant), using a CONF_PERCENT value of 10%, a CONF_MAX value of a day, and a CONF_MIN value of 0 for all URLs. We used the projected freshness lifetime durations. To simulate requests served by origin servers, we set the TTLs to be the respective projected lifetime durations. In various experiments we used TTL values lower than freshness lifetime to simulate access through a cache.

Since we could not GET all URLs in a reasonable time without adversely affecting our environment, we selected a subset. We used all URLs that were requested more than 12 times, and applied some weighted random sampling to others. In total we fetched about 224K distinct URLs (the original logs had millions, most of them requested only once). We factored the sampling back in by scaling the results and grouping URLs by number of requests.

Our simulations assumed infinite cache storage capacity, which is consistent with current trends and with the desired performance on the actual traces we used.

We used the *freshness rate* as our performance metric. We define the freshness rate as the ratio of freshness hits to content hits. We define *content hits* and *freshness hits* as follows. A request for a fresh cached object is considered a content hit and a freshness hit. A request for a stale cached object is considered a content hit only if an authority with a fresh copy (e.g, the server or another cache) certifies that it is not modified. We refer to requests that constitute content hits but not freshness hits as *freshness misses*. Content hit requests exclude requests to explicit non-cachable objects, requests on which the client specified to bypass caches (`no-cache` request header), and requests when the content had actually changed. We also excluded the first request for each object. Content hits include requests for objects with respective freshness lifetime value of 0.

We note that the freshness rate captures only one performance aspect of a cache and we later discuss how to interpret it and combine it with others.

The Squid logs recorded the response code returned to the client and the cache action. We only considered GET requests such that the cache returned a "200" or "304" response code to its client. We further classified these requests as follows, using the listed Squid labels.

- freshness hit (*fhits*): TCP_HIT, TCP_MEM_HIT, TCP_IMS_HIT
- freshness miss (*fmiss*): TCP_REFRESH_HIT
- content miss (*cmiss*): we separately accounted for
  - *cmiss-r* (the cache had a stale cached copy, issued an IMS request, and got a Modified response): TCP_REFRESH_MISS
  - *cmiss-d* (there was no cached copy): TCP_MISS
- `no-cache` request header: TCP_CLIENT_REFRESH_MISS

The following table shows the fraction of requests of each type.

log	fhit	fmiss	cmiss-d	cmiss-r	`no-cache`
UC	23%	10%	56%	1%	10%
SD	19%	15%	56%	3%	7%

Requests classified as fmisses, cmisses, or `no-cache` involve communication with the origin server or another cache. Freshness misses constitute 13% (UC) and 19% (SD) of all requests directed outside. The caches directed most validation requests (fmisses and cmisses-r) outside the NLANR hierarchy (100% in the UC cache and 99.3% in the SD cache). All these requests were directed to an origin server or a transparent reverse proxy. Our measurements suggested that the vast majority of requests directed outside reached the origin server. [4] The great exception was

[4]We distinguish a transparent reverse proxy cache from an origin server by issuing two consecutive GET requests within more that a few seconds apart to the same IP-address of the same server. A cache return a DATE header as obtained from the authoritative server. An authoritative server returns the current time according to its clock. Thus, we examined the DATE header value of the two HTTP responses. The same value indicated a cache. Two different values, where the difference approximates the time between our two HTTP GET request-response pairs, suggest origin server, very short TTL (few seconds or less), or a non-cachable object (that is, no age-penalty effect).

requests directed to CDN servers (this is discussed further in Section 5). Thus, by and large, the original trace was not subjected to age penalty. If more validation requests are directed to a cache we expect to see some of the freshness hits transform to freshness misses. It is also apparent that the vast majority (90% for UC and 95% for SD) of validation requests return Not-Modified.

4 High-level cache simulation

The following experiment attempts to measure the age penalty when content is fetched from nonauthoritative servers such as reverse proxy caches. As discussed above, a given request can constitute a hit or a miss at the higher-level cache. For request constituting cache misses, performance would have been better had the higher-level cache not been there, as longer latency and more traffic is incurred than through direct communication between our cache and the origin server. The higher-level cache is also not as effective on request constituting content misses. Thus, in order to distill the performance effects of age, we simulate an optimistic scenario where all requests constitute cache hits (the higher-level cache always keeps a fresh copy). Note that overall performance would be worse when this is not the case.

Our simulation corresponds to a situation where all requests to a given URL are forwarded from our cache consistently through the same top-level cache (e.g., a reverse proxy) [5]. These top-level caches maintain continuous freshness by refreshing copies through the respective origin servers as soon as it becomes stale. Under these assumptions, for objects with fixed freshness lifetime durations, the TTL of the copy at the top-level cache cycles from the lifetime duration to 0. We denote the freshness lifetime value by T and compute the TTL obtained after each (content or freshness) miss as

$$\text{TTL} = T - (\textit{time} - \textit{log\_start\_time}) \bmod T .$$

For requests labeled as CLIENT_REFRESH in the trace (arriving with no-cache request header), we set the TTL to T in order to simulate a situation where misses are forwarded to the origin server.

NLANR log & requests fraction	object source	freshness rate $fhits/chits$
UC 1	origin	52%
UC 1	cache	43%
UC 0.1	origin	35%
UC 0.1	cache	28%
SD 1	origin	47%
SD 1	cache	38%
SD 0.1	origin	30%
SD 0.1	cache	24%

Table 1: Freshness rates when requests are directed to (i) a cache or (ii) origin servers.

The respective freshness rates in the two scenarios are listed in Table 1, and show that the overall age penalty of going through higher-level caches amounts to 20%-25% decrease in freshness hits. The logs UC0.1 and SD0.1 are reduced traces that included a random sample of 10% of the requests.

5 Content Delivery Networks

Many sites now use content delivery networks (CDNs) to distribute some of their content [1, 10]. CDNs place multiple servers distributed inside different ISP's networks. Each of these servers is essentially a cache. Like reverse proxy caches, these servers only process URLs within the CDN domain, but like proxy caches, they are located close to clients. Since clients can reach a CDN server through fewer router hops and peering points, the response time, packet loss, and the number of data retransmissions typically improves. CDNs also off-load origin sites.

The current architecture used by popular CDNs (such as Akamai [1]) involves URL

[5]Interestingly, analysis shows that otherwise the age penalty is higher [3]

substitutions. The origin sites substitute the original object URL to one within the CDN domain. For example, the origin site substitutes the URL of the embedded image `http://cnn.com/images/icons/video.gif` with the URL `http://a388.g.akamaitech.net/7/388/21/e0a3b4215f9e4b/cnn.com/images/icons/video.gif`. The substituted URL is termed by Akamai the *Akamaized URL* or *ARL*.

Clients are then directed to a "good" CDN server (one that is likely to be less loaded, have a cached copy of the object, and be close to the client). When the client local DNS server resolves the hostname of the object, the Akamai DNS server returns an IP-address of a "good" CDN server. CDNs mainly serve relatively static content such as images and java applets but are striving to serve additional content types including streaming media, authenticated content and dynamic content.

The URL substitution architecture forces the deployment of some coherence protocol between the CDN and the origin sites. One natural possibility is to adhere to HTTP/1.1 freshness control - where copies on the CDN servers must be refreshed when they expire. This approach would make the aging issues similar to plain reverse proxy caches. We shall see, however, strong indication that a different mechanism is used.

Experimentation shows that the content of the response is not sensitive to the particular Akamai hostname used (e.g., when `a388.g.akamaitech.net` is substituted with `a534.g.akamaitech.net`). Furthermore, the response content is not sensitive to changes in the 3 fields of the ARL preceding the part that contains the origin URL (e.g., `/388/21/e0a3b4215f9e4b/` in the example above). Values in the last of these field suggest that it encodes a time duration, counter, or an object identifier. It seems that the redundancy in URL to ARL translation is used to encode freshness lifetime and that the Akamai servers use the information embedded in the requested ARL to determine when to refresh (through the original URL). In some cases (as in the example above) the ARL seems to be unique per version of the URL, as it encodes the version identifier (`e0a3b4215f9e4b`). Measuring latency when requesting different ARLs [6] strongly suggest that CDN servers refresh a copy when a request is received with a new "Akamization" of the same URL.[7] It is also evident that Akamai servers do not conduct any checks to determine if a particular "Akamization" was actually used by the origin site, since they responded to requests with arbitrary values (even in the URL portion of the ARL, e.g., in the ARL above, the URL portion `cnn.com/images/icons/video.gif` can be substituted with an arbitrary URL). With CDN servers not performing checks, freshness control can be performed either by (i) CDN servers considering a copy as expired after a pre-agreed duration had elapsed, or as the content of a particular embedded URL is modified at its origin site, the origin site has to (ii) re-substitute a new ARL for it or (iii) to trigger removal of all cached copies at all CDN servers. The most plausible possibility seems to be the first, that is, the use of pre-agreed durations, most likely encoded within the ARL itself.

We discuss how age affects performance under past and present CDN practices.

5.1 Leave response headers intact

The first practice we discuss was implemented by the two studied CDN sites until about March of 2000. The CDN servers, like plain HTTP caches should, populated the end-to-end header values of their responses with the original settings provided for the object when it was fetched from the origin site. As a result, when an object is fetched to a cache from a CDN server, its age includes the

[6]We conducted the following measurements with several URLs: We compared the latency of 500 GET requests of the URL from the same CDN server when (i) the same ARL was used each time (ii) a different ARL was used each time (e.g., by replacing the string `e0a3b4215f9e4b`). The cumulative latency in the latter measurement was 25%-40% slower than in the first. This suggests that when seeing a new Akamization, the CDN server obtains a new copy of the URL from the origin server prior to sending the response.

[7]As of 11/2000, Akamai seem to use HTTP redirect (302 response code) to the original URL in some cases when a new Akamization is seen.

duration it resided on the CDN server. More specifically, when a CDN server responded to an HTTP request it returned the DATE, EXPIRES, MAX-AGE, and LAST-MODIFIED response headers as they were when the object was fetched to the CDN server from the true origin site. The CDN servers, however, differ from HTTP caches since they are considered by caches to be the authoritative sources for the substituted URLs (ARLs). Hence, their responses are always considered valid for the present request (even if already expired) and they constitute a final destination of requests with `no-cache` request headers.

To experimentally measure the age penalty effect for CDN-delivered objects we extracted from the (January, 2000) NLANR logs requests that used two CDNs Akamai [1] and Digital Island (Sandpiper) [10]. To that end we extracted URLs whose hostname included the strings "akamai," "sandpiper", or had the prefix "fp.cache." We note that this is only a subset of requests served by these providers because sometimes the domain name does not include these strings. We used the same simulation methodology as outlined in Section 3. For these URLs we calculated TTLs in two ways: 1) According to the headers provided by the content provider, 2) As they would have been calculated if a current copy was obtained from the origin (with the DATE set to current time and adjusting for dynamically-generated EXPIRES headers.) Figure 4 shows the fraction of URLs with TTL value below x for varying x. When fetched directly from the origin, most objects have TTL value of a day (this is consistent with what we get when considering all objects in the log). When fetched from the CDN server, TTL values drop drastically, and most of them become zero.

The table below includes for each provider and cache, the respective fraction of requests in the log, the freshness rate as is (fetched from the CDN behaving as an HTTP cache with respect to the headers), and the freshness rate as would-have-been if the origin server was serving the object.

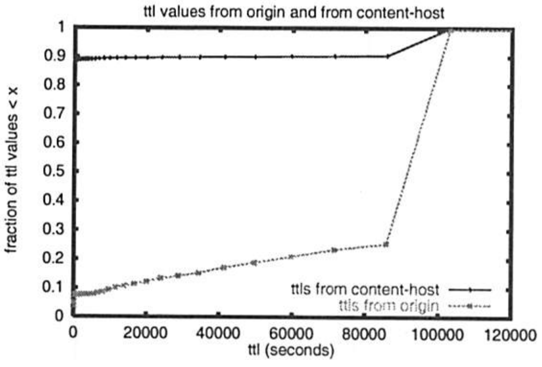


Figure 4: CDF of TTL values when content is fetched from origin vs. content-host

CDN	*cache*	*% of log requests*	*fhits/chits* thru: *CDN*	*origin*
Sandpiper	UC	0.4%	5%	76%
Sandpiper	SD	0.5%	6%	67%
Akamai	UC	1.7%	5%	61%
Akamai	SD	1.1%	6%	63%

These results show that requests to objects served through CDNs incur a ten to fifteen fold decrease in freshness hits and 2-3 fold increase in freshness misses relative to the same requests directed to respective origins.

Freshness misses constitute 26%-38% (UC and SD Sandpiper) and 26%-40% (SD and UC Akamai) of all requests forwarded by the UC cache to the CDN, which is considerable more than general statistics across all logged requests. Our simulation suggests that most of these requests are due to the age penalty effect.

What caused the age-penalty through CDN servers to be so much worse than through HTTP-compliant caches (see Section 4) was deployment of a freshness control mechanism other than HTTP/1.1 to decide when to refresh their copies of hosted objects. The freshness lifetime value used by the CDN server was typically considerably longer than the HTTP-implied value, and thus, the typical hosted object HTTP-expired well before it was refreshed by the CDN server. For example, often the content-host-served objects have an age exceeding MAX-AGE value, resulting in a TTL of 0 and them becoming stale almost immediately at an HTTP/1.1 compliant cache.

Around March, 2000, one of the CDNs, Akamai, addressed the issue by rewriting some of the end-to-end response headers of hosted objects. In particular, the DATE header value was set to the current time at the Akamai server, and the EXPIRES is made consistent with the MAX-AGE directive. The returned headers are then such that an HTTP/1.1 compliant cache assigns the same TTL as it would have if it had contacted the origin server[8]. This rewriting of response headers eliminated much of the unnecessary freshness misses. However, the interaction of the two freshness control protocols gives rise to further questions.

5.2 ARL freshness control

It is evident from the previous measurements that the freshness control mechanism between the CDN servers and the sites allows considerably longer freshness lifetime durations than HTTP freshness control. Thus, the freshness rate of client caches can be significantly improved if the freshness lifetime known to the CDN servers is translated into the HTTP response headers. In particular, when the "Akamization" is unique per version, the EXPIRES header can be set accordingly to a far time in the future, and as a result, greatly reduce the number of freshness misses at downstream caches and network load due to validations.

To asses the potential gain we used a 5 day long trace from the UC NLANR cache taken between 25 and 29 of July, 2000. Note that this trace was downloaded after Akamai implemented the change of rewriting response header values. The trace included for each request the response code returned from the UC cache to the client, the UC cache action taken to process it, and service times (the total processing time from the UC cache perspective).

In total there were 19K requests to 6K different ARLs in the akamaitech.net domain for which the UC cache returned a 200 (OK) or 304 (Not-Modified) response code to the client. 304 responses are freshness misses at the client cache and 200 responses are content misses at the client cache. The breakdown of requests is given in Table 2. In each category, we further breakdown requests according to actions taken at the UC cache. In particular, we list the percentage of freshness hits at the UC cache (LOCAL), misses forwarded to a sibling cache (SIBLING), and misses forwarded to the Akamai server (DIRECT)[9]. All request directed to a SIBLING were cache content misses (TCP_MISS in Squid terminology). For DIRECT requests we accounted separately for those that arrived from the client with a no-cache request header (TCP_CLIENT_REFRESH_MISS), those that were freshness misses at the UC cache (TCP_REFRESH_HIT), and other cache misses. We calculated average service time for requests in each category excluding values exceeding 10 seconds, since otherwise the average is distorted by few outliers. Service times of less than 10 seconds included all local responses and about 99.5% of all responses. It is evident that average service times highly varies by request category. LOCAL requests are handled significantly (70%) faster than requests on which the cache contacted another server. The response time on 304 responses is about 40% shorter than on 200 responses.

We are now able to asses the potential reduction in freshness misses, both at the UC cache and at caches of its clients. 10.4K out of the 19.4K (over 50%) of the requests directed to the UC cache from client caches were validation requests with 304 response. All of these requests except for those with a no-cache request header could have been eliminated. Thus, 45% of total requests between client caches and the UC caches would have been eliminated. Considering average service times shows that the average latency gain would have amounted to about 150ms (plus RTT) per request. Requests from the UC cache to Akamai servers that potentially could have been eliminated are all validation requests without a no-cache request header. This included about 19% (11% for 200 client

[8]This is true in all cases except when the EXPIRES header is used in a relative way and there is no MAX-AGE directive (see Section 2), and when clocks on the origin site and the Akamai server are not synchronized

[9]no requests were forwarded to a parent cache

at client cache	*at UC cache*				
content miss (200)	*LOCAL*	*DIRECT*			*SIBLING*
9K 208ms	35% 80ms	59% 269ms			6% 360ms
		no-cache	304	*miss*	
		20% 1.1K	12% 0.62K	68% 3.6K	
freshness miss (304)	*LOCAL*	*DIRECT*			*SIBLING*
10.4K 157ms	28% 51ms	67% 196ms			5% 201ms
		no-cache	304	*miss*	
		18% 1.3K	29% 2K	53% 3.5K	

Table 2: Breakdown of requests according to cache action.

responses and 29% for 304 responses) of total requests issued to an outside server (sibling cache or origin).

On a side note, it is evident that the UC cache evicts items more aggressively than its clients caches: the majority of requests where 304 response was sent to the client were content misses at the UC cache and on the other hand 60% of the freshness misses at the UC cache were freshness misses (and content hits) at the client. This suggests that UC cache performance on these items (and the possible gain from ARL uniqueness) would benefit from increased storage or better replacement policy.

In an attempt to try to consider only ARLs that are used in a unique fashion per URL version we considered only requests to ARLs where the value in the last field preceding the URL seemed to contain a time-stamp, counter, or version identifier. We identified 5.8K such requests which included 2.9K with 304 response and 2.9K with 200 response. Out of the 304 responses there were 0.8K with no-cache headers. Thus, about 35% of total requests from clients to the UC cache would have been eliminated. Out of the 5.8K requests, about 60% were DIRECT and about 13% were DIRECT freshness hits. Thus, about 20% of requests to the Akamai server could have been eliminated.

The above discussion asses the potential gain from translating the proprietary CDN freshness control into the HTTP headers, but leaves aside possible reasons for keeping two sets of directives. For example, hit-count and collecting statistics, but these can usually be performed through the referring HTML page or using a single embedded object on each page.

6 Conclusion

We discussed the effects of object-age on the performance of cascaded caches, and showed, through trace-based simulations, that the age penalty can be significant. We remark that the age penalty can be eliminated altogether if strong consistency is maintained between high-level caches (e.g., reverse-proxies or CDN servers) and origin servers. Strong consistency, however, is expensive and not facilitated through HTTP or otherwise widely supported.

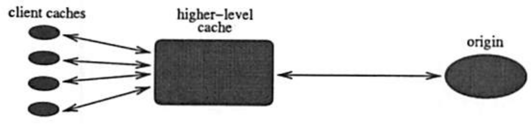


Figure 5: Client caches, high-level cache, and the origin

For future work, we propose two approaches to alleviate the age penalty while working within the current consistency infrastructure. The first, *source selection*, is deployed by "low-level" caches. In some architectures a cache may have a choice of where to forward requests on which a miss occurred. If so, it is preferable to forward requests to a server that is more likely to have the "youngest" fresh copy of the object. More generally, source selection should balance distance, likelihood of fresh cached copy, and age. The second approach, *rejuvenation*, is deployed by "high-level" caches. We use the term *rejuvenation* for pre-term validation of selected copies (well before they expire), as a mean to decrease age. At the limit, frequent rejuvenation amount to strong consistency between the cache and the origin and thus, no age penalty. Rejuvenations reduce traffic between the cache and its clients (and user-perceived latency), but increase traffic between the cache and the origin servers (see Figure 5). When the cache serves many clients which request an object frequently, the benefit of decreased age penalty could significantly outweigh the cost. We analyse and experimentally-evaluate rejuvenation in subsequent work [4, 3].

The age penalty can widely vary for content served by CDNs. The practice of intact end-to-end HTTP response headers resulted in an order of magnitude decrease in freshness hits and in a 2-3 fold increase in validation traffic. With edited response headers, it seems that performance could greatly improve if ARL freshness control is translated to HTTP freshness control in the rewritten HTTP response headers. Generally, it is desirable that CDN clients would need to use only one set of freshness control specifications, either through HTTP headers, or through the CDN proprietary mechanism. CDNs can then either adhere to HTTP directives or re-write them to be consistent with the proprietary protocol. A related discussion on defining the role of CDNs, possibly by incorporating specific CDN directives into HTTP headers, is given in [9].

We conclude with the hope that our work would contribute to better understanding, by researchers and practitioners, of the age-related facet of cache performance, and ultimately, spur further work aimed at improving performance of cascaded caches.

Acknowledgment

We thank Duane Wessels for answering questions on a (since corrected) Squid logging bug, and Bruce Maggs and Mark Nottingham for answering questions on observed Akamai practices. We also thank Jeff Mogul for comments on an earlier version of this paper.

References

[1] Akamai. `http://www.akamai.com`.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol — HTTP/1.0. RFC 1945, MIT/LCS, May 1996.

[3] E. Cohen, E. Halperin, and H. Kaplan. Performance aspects of distributed caches using TTL-based consistency. Manuscript, 2000.

[4] E. Cohen and H. Kaplan. Aging through cascaded caches: performance issues in the distribution of web content. Manuscript, 2000.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, and T. Leach, P. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, ISI, June 1999.

[6] A Distributed Testbed for National Information Provisioning. `http://www.ircache.net`.

[7] J. C. Mogul. Errors in timestamp-based HTTP header values. Technical Report 99/3, Compaq Western Research Lab, December 1999.

[8] M. Nottingham. Optimizing object freshness controls in Web caches. In *The 4th International Web Caching Workshop*, 1999.

[9] M. Nottingham. On defining a role for demand-driven surrogate origin servers. In *The 5th International Web Caching and Content Delivery Workshop*, 2000.

[10] Digital Island (Sandpiper). `http://www.sandpiper.com`.

[11] Squid internet object cache. `http://squid.nlanr.net/Squid`.

On Line Markets for Distributed Object Services: the MAJIC system

Lior Levy, Liad Blumrosen and Noam Nisan*

Institute of Computer Science,
The Hebrew University of Jerusalem,
Jerusalem, Israel.

Abstract

We describe a general-purpose architecture for applying economic mechanisms for resource allocation in distributed systems. Such economic mechanisms are required in settings such as the Internet, where resources belong to different owners. Our architecture is built above standard distributed-object frameworks, and provides a "market" for arbitrary distributed object resources. We first describe the abstract elements and properties of an architecture that can be applied over essentially any distributed object-based platform. We then describe the MAJIC[1] system that we have implemented over Suns' Jini platform. A key novel aspect of our system is that it handles multiple parameters in the allocation and in the specification of utilities and costs for each distributed service. We provide both theoretical and experimental results showing the following three key properties of this system: (1) Efficient resource allocation. (2) Motivation for resource owners to share them with others. (3) Load balancing.

1 Introduction

1.1 Motivation

The following concept may be viewed as the holy grail of "Internet Computing": Every user connected to the Internet should have complete access to all resources available anywhere on the Internet. The user should be presented with an illusion of a single centrally organized "global computer". The main challenge of computer engineering, in this context, is to design the protocols, algorithms, paradigms, and systems that achieve this illusion by using the aggregation of the physical computers, communication links, and other resources that are available on the Internet.
Ideally, such systems would optimally allocate all available resources across the Internet. There are indeed a wide variety of such resources: computational resources (such as CPU time, or file servers), information resources (Databases, video), communication resources (links, QoS), services (help-desk, access to specialized algorithms), hardware (printers, cameras), and more. A dream suite of protocols and algorithms for the Internet would allow all these resources, as well as others, to be optimally and transparently allocated across the Internet.
There are clearly many aspects that need to be addressed on the way to this "holy grail", and many of these aspects have received much attention in the literature. In this paper we concentrate on the interplay and synergy between two key paradigms that address different aspects of this challenge: the distributed objects paradigm for basic technological interoperability and the economic paradigm for motivating resource sharing by different users or organizations.

1.2 Distributed Objects Paradigm

In recent years the paradigm of distributed object services is becoming the basic backbone of communication and cooperation between components of a dis-

*Email: {liorl,liad,noam}@cs.huji.ac.il. Supported by grants from the Israeli Ministry of Science and the Israeli Academy of Sciences.

[1]**M**ultiparameter **A**uctions for **JI**ni **C**omponents

tributed system. In a distributed object framework, computers on a network encapsulate their shareable resources (services) in well defined procedural interfaces. Other computers then use these resources by performing remote procedure calls (RPC), or in an OOP terminology, remote method invocations (RMI) on them. In its pure object-oriented variant this paradigm is the basis of most modern commercial distributed platforms: CORBA [5], Microsoft's DCOM [6], Java's RMI [35]. Taking a wider perspective, web servers follow this paradigm on the level of web pages (static or dynamic), and with standards such as XML [49] and protocols such as SOAP [41], a true web-like infrastructure for distributed processing that follows this paradigm emerges. Indeed many authors have proposed variants and implementations of this vision under names such as "Web of Objects", "Distributed Objects Everywhere", etc [33, 48, 42, 13, 3].

1.3 Economic Paradigms for Resources Sharing

A major difficulty in achieving efficient sharing of resources across the Internet is the obvious fact that the different computers and resources belong to different organizations. An Internet-wide resource sharing system must provide motivation for the owners of resources to share them with others. Any such motivation leads to some kind of economic system, and in its simplest form involves payments for services. Such economic systems for distributed allocation of computing resources have been applied to CPU time [9, 47, 46, 45, 27, 34], communication [20, 40, 37, 19], and other resources [21, 15, 38, 45], and have received much theoretical interest lately [20, 25, 10, 32, 16, 39, 28, 29]. Such systems pursue two complementary goals: that participants are indeed motivated to share these resources with others and that the resources are indeed allocated well.

1.4 This Paper

We first propose a general architecture for augmenting a distributed-object system with payments, and a market-based mechanism for allocating resources. We then describe a system of this sort, the MAJIC system, that we have implemented over Sun's Jini infrastructure. More details are available in the MAJIC web site [22]. This architecture allows, for the first time, applying the ideas of economic-based cooperation to the full spectrum of resources available on the Internet: CPU time, file servers, Databases, online entertainment, communication bandwidth, algorithms, printers and other hardware, etc. Moreover, this is done in a way that is easily inter-operable with the current leading technologies. We provide both general arguments (and mathematical proofs) showing that such an augmented system would indeed function well, and specific experimental results from MAJIC, supporting these findings.

This architecture provides, for the first time, an economy based general-purpose infrastructure for all kind of resources, compared with earlier similar systems, which were dedicated to one kind of resource. A key novel aspect of our architecture is that it allows multiple parameters in specifying utilities and costs for each service request and handles these parameters in an efficient and non-trivial way.

It is clear that any system that achieves serious resource sharing over the Internet must address both the issue of technical inter-component communication and the issue of motivating selfish entities to share their resources. We believe that systems built along the principles laid here answer both issues in an integrated, inter-operable, and efficient way and could provide a general-purpose architecture that allows true efficient resource sharing on the Internet!

2 A blueprint for a market of distributed object services

2.1 Basic idea

The starting point of our architecture is simple: each object that provides a service may attach a "price-tag" to it. When another object wishes to use a particular type of service, it calls a central "service market place" that functions as the object request broker. The market performs a "reverse auction" for this service, where all available objects (service providers) that provide a service of this type can participate. The provider that charges the lowest price wins, and gets to service the request for the agreed-upon price.

A simple example that we will use throughout the paper is a printing service. Our starting point assumes that printers provide the service of printing a page by providing the simple remote method: *printer.print(page)*. Each printer has a price for this service: *printer.getPrice()*. A computer that wants to print a page on a printer that does not belong to him gets the reference to the cheapest printer from the "printing market" and can then print on this printer: *market.getPrinter().print(page)*.

Several basic issues need to be addressed before this can be made into a workable system, and we describe the major ones.

2.2 Parameterized Services

Looking at the previous printing example, one is immediately concerned with the differences between different printers and printing jobs. In reality many parameters distinguish one print job from another: number of pages, page size, printer's location, printing speed, print quality, etc. Clearly any serious system that handles printing services must be aware of these differences. More generally, distributed object services receive input parameters - it is quite clear that the price requested for a service must be tightly related to these parameters. Additionally, inter-changeable distributed service providers are still not totally equivalent in many of their parameters (such as their quality, virtual location or their speed).

This is not a simple issue to tackle when one attempts to produce a "market" for these services. Ignoring the parameters in the market will simply make any type of efficiency in resource allocation impossible. Taking all the parameters into account in the definition of the "market" will lead to a logically separate market for each request and for each service provider, eliminating competition and thus any flexibility in allocations. The solution is clearly to work within a single market, but take parameters into account during resource assignment.

In our system each *service type* specifies the set of parameters of the service, defining the *parameter space* of the service. Each service provider can supply the service in some specified subset of the parameter space. Back to our example, a certain printer may only be able to print on "A4" or "letter" page sizes, but not on "A3" size, while another may also offer "A3" size. Similarly, a printer will usually only supply the printing service at a certain physical location or with a certain time delay (depending on its current load). Each service request may be in a single point of the parameter space ("I need A4 pages"), or may be satisfied with a whole subset of the parameter space ("A4 or letter is fine"), possibly with preferences among the different possibilities.

In economic terms, each point in the parameter space of a service type is a separate product type. The products that correspond to different points in the parameter space of a single service type are *partial substitutes* to each other - both for service providers and for service requesters. The challenge we face (and solve below) is to organize a single market for all these products, while handling the partial substitution in an economically efficient way.

2.3 Sellers' Quote and Buyers' Utility

Our economic system is based on a common currency in which all participants can express their economic preferences. Thus we assume that each service provider - *seller* - has a certain internal cost for supplying the service at each point of the parameter space that can be provided by him. Similarly, each service requester - *buyer* - has a certain economic benefit from receiving the service, a benefit that may depend on the parameters. Denote by P the parameter space of a certain service type, our economic model of participants is given by the following two functions: (1) Sellers' *cost* function: $c_S : P \to R^+$. For a point $p \in P$, $c_S(p)$ specifies seller S's internal cost for supplying the service with parameters p. I.e., he would like to provide this service with these parameters if he is paid more than $c_S(p)$, and would not agree to provide the service for a lower price. We take the convention that if S cannot provide the service with parameters p at all then $c_S(p) = \infty$. (2) Buyers' *utility* function: $u_B : P \to R^+$. For a point $p \in P$, $u_B(p)$ specifies buyer B's benefit from receiving the service with parameters p. I.e., he would like to receive this service with these parameters if he pays less than $u_B(p)$, and would not agree to buy the service for a higher price. We take the convention that if the service with parameters p is not acceptable at all to him, then $u_B(p) = 0$.

In our system, each seller sends to the market a function corresponding to his cost function – called the *quote* function $q_S : P \to R^+$. For a point $p \in P$, $q_S(p)$ specifies his "quote" for the service with parameters p – i.e. the amount of money he demands for providing the service with these parameters. The quote function q_S is essentially a catalog specifying a price for each choice of parameters that can be supplied by S. Informally speaking, the quote function q_S of a seller should correspond tightly to his cost function c_S. This however cannot be guaranteed at this point as a seller will likely send to the market a quote function aimed for maximizing his profits - not aimed at any particular correspondence with his cost function. We will return to this point below. The system allow sellers to modify their quote functions occasionally, taking into account changes in status.

When a buyer requests a service he sends to the market a representation of his *utility* function. The market matches this buyer with the seller that fits him best. The abstract process that takes place is that the market creates an agent for the buyer based on his *utility* function. This agent is presented with the catalogs (*quote* functions) of all sellers, and chooses the best seller and parameters. The details of this process are explained in section 2.4.

The representation of the *quote* and *utility* functions as objects may be given by specifying the formulas that define them as a function of the different parameter values or may be given by opaque distributed objects that encapsulate these functions. The choice of representation will usually depend on the service type and has consequences in how the system can function.

2.4 The Market Mechanism

As mentioned, the market holds the current quotes from all sellers $\{q_S\}$, and when it receives a request from a buyer - specified by the buyer's utility function u_B – it attempts matching this request to the best seller and choosing the best parameter values. The optimization criteria is the *surplus*: $surplus_{B,S}(p) = u_B(p) - q_S(p)$. For a given supplier S, the parameters p are chosen as to maximize this surplus: $p^*_{B,S} = \arg\max_p\{surplus_{B,S}(p)\}$. In case this search over the parameter space is computationally feasible (this depends on the representations of the parameter space and the quote and utility functions), the buyers' agent can directly find these optimal parameters. Otherwise, the system allows each buyer to supply a "parameter search engine" that attempts finding these parameters, given a quote function (encapsulated by an object) as input. We expect this mechanism to be computationally efficient in many cases, either because of the simplicity of the parameter space, or because of the small number of parameters. However, when searching in complex parameter space, we expect the buyer's search engine to find a close approximation in reasonable time.

The optimal supplier is chosen as to optimize the surplus under the optimal parameters: $S^* = \arg\max_s\{surplus_{B,S}(p^*_{B,S})\}$. At this point the buyers' agent must check that this surplus is indeed positive: $surplus_{B,S^*}(p^*_{B,S^*}) > 0$. Otherwise, the buyer is not willing to pay as much as the optimal seller is asking, and the service request should be canceled. As analyzed theoretically in section 4, and demonstrated experimentally in section 5, this agent-based allocation produces efficient allocations in terms of reported utilities and quotes.

Once the optimal $S^*$and $p^*$are found, the market may in principle fix any payment d in the range $q_{S^*}(p^*) \leq d \leq u_B(p^*)$. Any price in this range will be acceptable both to the buyer and to the seller. The simplest choice would be to use the quote function as the price: the buyer must pay the seller the amount of $q_{S^*}(p^*)$. This is certainly the usual choice in commerce as it corresponds to the catalog price of the chosen product. In terms of auction theory, this corresponds to a first price auction [31, 24].

We suggest also a different choice of payment, generalizing Vickrey's second price auction [44]. The motivation for this payment rule is to motivate sellers to send the market a quote that is equal to their cost function $q_S = c_S$ – a property known as *incentive compatibility.* This is extremely important due to the fact that otherwise the assignment does *not* optimize the allocation according to the true costs but rather according to the quotes. Indeed the previously mentioned payment rule motivates sellers to announce a quote that is higher than their costs – thus potentially leading to a wrong choice of seller. For general background on this topic see [14, 4, 23], and for specific discussion in the context of computation resources see [28, 26, 36, 43].

The payment rule we suggest is as follows. Let S^2 be the second best choice for supplier: $S^2 = S^2_B = \arg\max_{S\neq S^*_B}\{surplus_{B,S}(p^*_{B,S})\}$. The surplus with supplier S^2 is less or equal to the surplus with supplier S^*. This payment method mandates that the buyer only gets the surplus of S^2, while the optimal seller gets the difference between the two surpluses. Thus the payment to supplier S^* is given by: $d = u_B(p^*{}_{B,S^*}) - surplus_{B,S^2}(p^*{}_{B,S^2})$. The main theoretical result we show, using the standard game-theoretic models of rational behavior, is that this payment method results in incentive compatibility, and thus all rational sellers indeed quote their true costs leading to efficiency of allocations in the system.

2.5 Load Balancing as a By Product

As described above, this type of system ensures optimal allocation of each request to the service provider that is best for it. In cases that requests do not conflict with each other, this implies that the system obtains optimal global performance. This is clearly not the usual case! The whole point of allocation in distributed systems is handling the conflicts – different requests should normally be split between the available servers. Going back to our printing example, not everyone can gain access to the best and cheapest printer – this would likely cause a bottleneck there. Indeed, perhaps the most basic requirement from a distributed system is that of load balance: the load should be reasonably split between available servers.

A key observation is that economic-based systems can provide this load balancing – if designed correctly. Specifically, when one considers the underlying reason why load balancing is usually desired, it seems that the reason is simply that users want their requests to be served quickly. Putting this into economic terms, users' utilities depend on the time until their request is serviced. This is rarely formalized, but may be easily formalized if service-time is a parameter in the parameter space of the service type – as it is in our system. E.g., a request may specify a firm deadline by setting the utility to be zero if the service-time is greater than the required deadline. Similarly, gentler penalties for tardiness may be applied by tailoring the utility function's dependence on time. Suppliers, on the other hand, must make sure that their quotes do indeed reflect their current capabilities in terms of service-time and must modify them when their load changes.

Load balancing emerges automatically once the quotes of the different suppliers do indeed reflect their actual service-time capabilities. As a certain service supplier gets more requests assigned to it, it must raise the service-time promised in his quotes. This will automatically cause time-sensitive requests to be allocated to other service suppliers – those with lower loads. Requests that are less sensitive to service-time and more sensitive to other parameters that are optimized by a loaded supplier may still be assigned to it. This form of load balancing strikes a balance between optimized matching of service parameters and reducing the service-time in a way that is *locally* perfect: exactly according to the specification of the service request.

This local optimization does not necessarily imply global optimal balance of load: an assignment of a certain supplier to one request may result in a heavy penalty for the next request. Indeed, any formalization of optimal global allocation is computationally intractable (NP-complete) [11], and moreover, cannot be done in an online mode – servicing requests as they come [8]. Yet, we supply theoretical evidence as well as experimental results, suggesting that load balance emerges.

3 The MAJIC system: Multi-parameter Auctions for JIni Components

The MAJIC system is built on top of Sun's Jini platform, while implementing the basic architecture described above. We chose the Jini platform since it is a simple yet powerful distributed object system with open source. Moreover, Jini's object-broker mechanism – the *lookup* service – turned out to be easily adapted to our purposes. In addition, Jini uses Java's code mobility capabilities, which in our case allows transfer of utility functions and quote functions encapsulated in objects.

3.1 Jini overview

The *Jini*^{TM} system is a distributed systems technology released by Sun Microsystems in 1999. The Jini technology enables all types of digital devices to work together in a community, without extensive planning or installation. It is built on top of the Java environment [17] and the RMI mechanism [35]. Detailed specifications and explanations regarding the Jini system can be found in [1, 7]; on-line documentation can be found in [18]. We now describe only the bare essentials that are directly required for our purposes.

The Jini technology infrastructure provides mechanisms for devices, services, and users to join and detach spontaneously from a network, and be visible and available to those who wants to use it. Each Jini system is built around one or more *lookup services.* A *lookup service* is a service that maintains a list of known services and provides the ability to search and find services via the *lookup* protocol. When a service is booted on to the network, it uses the *discovery* protocol to find the local lookup service. The service then registers its proxy object (a Java object) with the lookup service using the *join* protocol. When a client program queries the lookup service for a particular service (using the *lookup* protocol), the lookup service returns the appropriate service proxy (or a set of service proxies) to the client. Then, the client can invoke methods of the chosen service using the proxy. The invocation can be done either locally or remotely (using Java *RMI* protocol). Figure 1 illustrates the system normal flow.

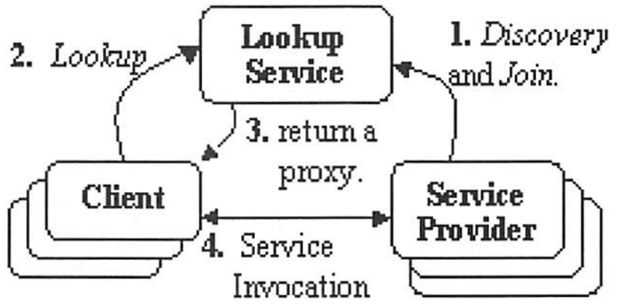


Figure 1 - Jini protocols flow

We use two other Jini concepts inside MAJIC; the first is the *leasing* mechanism ([7] ch. 10), which provides Jini its self-healing nature. This mechanism is a timed-based resource reservation: if a service fails or stops (either intentionally or unintentionally) without "cleaning up" after itself, its leases will eventually expire and the service will be deleted. The second concept is the service attribute set ([7] ch. 7), a flexible way for services to annotate their proxy objects with information describing the service; thus helping clients to find their required services.

3.2 MAJIC Architecture overview

The MAJIC architecture is based on the general blueprint described above applied to Jini platform. The main considerations were to preserve interoperability and the programming paradigms of the original Jini platform, while providing maximal flexibility. In addition, the market mechanisms should have minimal effect on the performance of the system.

3.2.1 Service Types

Each service type in our architecture is implemented as a *Jini service* that additionally implements the *Economy Service Interface* (ESI). Each service type has a well known set of parameters that defines the *parameter space*. The parameters are partitioned into *seller parameters*, those that are totally fixed by each seller, and *buyer parameters* which can be chosen by each buyer. Each service type defines a *Service Contract* class (a subclass of the abstract SC class described below) for sellers and a *Buyer Valuation* class (a subclass of the abstract BV class described below) for buyers.

3.2.2 The market

The *market* is implemented as an extension (subclass) of Jini's *lookup service* that uses economic mechanisms to perform efficient allocation. When service providers (sellers) *join* the market, they submit their *quote* wrapped in a *Seller Contract* (SC) object, passed as one of the *service attributes*. Whenever a buyer performs a *lookup* using the market, the buyer submits his *utility* function and its *parameter search engine*, both wrapped in a *Buyer Valuation* (BV) object. This is used by the market to create a buyer agent that performs the economic search for the optimal seller and parameters. Finally, the market returns a proxy to the optimal seller as the result of the lookup request. Additionally, a *Final Contract* object encapsulating the closed deal is created and sent to the buyer and to the seller. We have implemented two distinct markets, corresponding to the two payment methods described in the blueprint (first or second price).

3.2.3 The seller

The seller is a Jini service provider (thus additionally implementing the ESI interface). When joining a market, the seller should submit its *quote* function encapsulated in the SC object. Note that the quote function may be implemented as an arbitrary Java method; the method's code is actually transferred to the market. In addition to the quote function, the SC object also includes the fixed values of all seller parameters as well some timing-control information to be described below. Using the ESI interface, the seller receives online notifications of all *Final Contracts* (described below) and must be able to update and resubmit its SC object to the market. When the seller is actually invoked by a buyer using a previously obtained proxy, it must verify the validity of the corresponding FC.

3.2.4 The buyer

The buyer is a simple Jini client that in a lookup request sends the market its BV object. The BV object encapsulates both the *utility* function and the *parameter search engine* that finds the best values for the *buyer parameters* (accessed through a *getBuyerParameters()* method*)*. Both of these may be implemented as arbitrary Java methods whose code is transferred to the market and used as the buyers' agent. The parameter search engine may use well-known structure information regarding the parameter space of a particular service type, or may perform an exhaustive search of the parameter space.

3.2.5 Final contract

The *Final Contract (FC)* is a sealed contract that represents a closed deal between a buyer and a seller; it is constructed by the market and contains the following: a unique identifier, the SC, the chosen *Buyer Parameters* (BP), and the payment details. The FC should, in principle, be digitally signed by the market (this is not currently implemented) and is to be used as the basis for the actual electronic fund transfers.

3.2.6 Time-control mechanisms

Two mechanisms are supplied for ensuring that sellers' time-dependent quotes are used only if they are up to date. Every SC that is submitted to the market allows specifying a maximum number of contracts that may be created according to this SC. Whenever the maximum is reached, the market removes this seller from its list of service providers. Sellers with time-dependent quotes can specify a low maximum and then must periodically update their SC prior to this maximum being reached. In addition, each FC contains a lease control object (LCO) that ensures sellers that their services will be invoked by buyers within a predefined time interval. When the lease expires, the service proxy can no longer be used and an exception is raised.

3.3 Participants obligations

There are some implicit contracts (commitments and obligations) between all entities (players). The most significant assumption is that the market is *trustable* and *accepted* by all sides.

The market can assign buyers to a seller, as long as its SC is *valid.* The SC is *valid* as long as the service is registered at the lookup service and the seller hasn't supplied the maximum number of contracts mentioned in his SC. The seller must provide the service according to the parameters published in the FC, as long as the FC is *valid (*i.e. the FC lease hasn't expired). The seller is required to change all necessary parameters inside its SC whenever relevant (time dependent parameters). The buyer can execute the service during the lease duration, defined in the FC. Both seller and buyer accept the payment details described inside the FC.

4 Theoretical Analysis

We describe a theoretical model that analyze our multiparameter market system.

4.1 The Model

The Model is based on two types of players: a buyer and a seller, and a market place. Denote by P the parameter space of a certain service type.
Each Seller S, has:

1. A private *cost* function: $c_S : P \to R^+$. If S cannot provide the service with parameters p, then $c_S(p) = \infty$.
2. A public *quote* function $q_S : P \to R^+$. This function is sent to the market.

Each Buyer B, has two functions that are coupled together:

1. A *utility* function: $u_B : P \to R^+$. If the service with parameters p is not acceptable at all to him, then $u_B(p) = 0$.
2. A function $g_B : Q \to P$, where Q is the space of all possible *quote* functions. This function acts as the *parameter search engine* that finds appropriate parameters, given a *quote* function.

See section 2.3 for explanations.
Definition 1. g_B is called *optimal* if $\forall q_S,\ g_B(q_S) \in \arg\max_{p \in P}\{u_B(p) - q_S(p)\}$. I.e., g_B finds the parameters that maximize the buyer *surplus.*

4.1.1 The assignment mechanism

The market holds the current *quotes* from all sellers $\{q_S\}$, and when it receives a request from a buyer B, (u_B, g_B), it attempts matching this request to the best seller and choosing the best parameter values. The optimization criteria is the *surplus*. For a given supplier S, the parameters p are chosen by the buyer's *parameter search engine:* $p^*_{B,S} = g_B(q_S)$. The optimal seller is chosen as to optimize the surplus under these parameters: $S^* = \arg\max_s\{surplus_{B,S}(p^*_{B,S})\}$. If $surplus_{B,S^*} \leq 0$ then the request is denied. Otherwise, the market fixes a payment d_{B,S^*} according to one of the following payment methods:

- *first price:* $d_{B,S^*} = q_{S^*}(p^*_{B,S^*})$
- *second price*:
  let $S^2 = \arg\max_{S\neq S^*}\{surplus_{B,S}\}$.
  $d_{B,S^*} = u_B(p^*_{B,S^*}) - surplus_{B,S^2}(p^*_{B,S^2})$
  If $surplus_{B,S^2} \leq 0$ or S^2 does not exist then $d_{B,S^*} = u_B(p^*_{B,S^*})$.

Finally, the market outputs the assignment $B \longleftrightarrow S^*$ and the payment d_{B,S^*}.

We assume the following: (1) Several sellers have registered at the market and can alter their *quote* functions at any moment. (2) Buyers arrive to the market online and immediately receive a seller assignment.

4.2 Model Properties

We show that this model has the following properties: incentive compatibility, allocation efficiency and load balancing. Our analysis uses game theoretic notions that are standard in the field of mechanism design (see [23, 30]). Proofs for all theorems can be found in the full version of the paper [22].

Definition 2. (seller gain) For a fixed buyer (u_B, g_B) the gain of seller S is

$$gain_S(q_S, q_{-S}) = \begin{cases} d_{B,S} - c_S(p) & B\,gets\,S\,with\,p \\ 0 & otherwise \end{cases}$$

where q_{-S} is a vector of the quotes of all other sellers.

Definition 3. (Dominant strategy) A strategy (*quote*) q_S of seller S is called *dominant* if for every other declared quote $\widehat{q_S}$ and for every declarations of all other players q_{-S} , $gain_S(q_S, q_{-S}) \geq gain_S(\widehat{q_S}, q_{-S})$.

Definition 4. (Incentive Compatibility) A market mechanism will be called *incentive compatible* if declaring the true cost function ($q_S = c_S$) is a dominant strategy for all sellers.

Note: We do not discuss in this paper incentive compatibility for buyers as it is known that this can not be achieved concurrently with incentive compatibility of sellers as known from the analysis of bilateral trade [23].

Theorem 4.1. *Assume that (1) the assignment of services does not cause changes in any c_S and (2) g_B is optimal for all buyers, then the second price MAJIC mechanism is incentive compatible.*

Remark. According to [29], when g_B is not optimal, the VCG mechanism is not incentive compatible. Nevertheless, there are methods that achieve feasible approximation to incentive compatibility; such methods are described in [29].

Lemma. *Under assumptions of theorem 4.1,* $\forall u_B \forall \widehat{q_S} \forall q_{-S}\ \ gain_S(c_S, q_{-S}) \geq gain_S(\widehat{q_S}, q_{-S})$.

Definition 5. The total welfare achieved by the market is $\sum_{B\ gets\ S\ with\ p.}(u_B(p) - c_S(p))$.

Theorem 4.2. *Assume that (1) for all sellers $q_S = c_S$; (2) for each buyer g_B is optimal; (3) the assignment of services does not cause changes in any c_S. Then, for every sequence of service requests the total welfare is optimized by the market's allocation.*

We can show that the *load balancing* achieved by this economic-based model is good in many situations. Specifically, if all service suppliers are identical, then we would expect for almost uniform allocation of work among the suppliers.

Theorem 4.3. *Assume that (1) All the service providers are identical; (2) All buyers place a positive utility on faster service-time; (3) All quotes are correctly updated to reflect to the service suppliers true load. Then, the allocation obtained by the market achieves a makespan (last completion time of a service) that is within a factor of 2 w.r.t. the optimal allocation.*

It is shown in [2] that no algorithm can achieve a better competitive ratio. As usual, and as demonstrated by our experiments, typical behavior is much better and applies in a wider class of situations.

5 Experimental results

We have performed several kinds of tests on the MAJIC system: the system performance overhead, the load balancing effect, and the resource allocation efficienc. We have created two types of services and corresponding clients: a trivial service, called *Simple*, and a complex one, called *Printer*. The simple service has a single parameter (price) and the corresponding client has a fixed utility function. The printer service has several parameters: price per page, service-time (the time that the client request can be served), quality and speed. The service-time parameter varies with time to achieve simulation of time dependent services.

5.1 Testing environment description

We have built a network based testing environment that enables us to execute services and clients on several machines simultaneously. The entities (lookup service, services and clients) and their parameters can be externally configured. We have used a single machine (PIII-600 processor, 128KB RAM, WinNT 4.0 OS) for invoking the lookup service and the service providers, and another machine (PIII-600 processor, 2GB RAM, Linux OS) for invoking the clients. The machines were connected by LAN (Ethernet 10Mb/s).

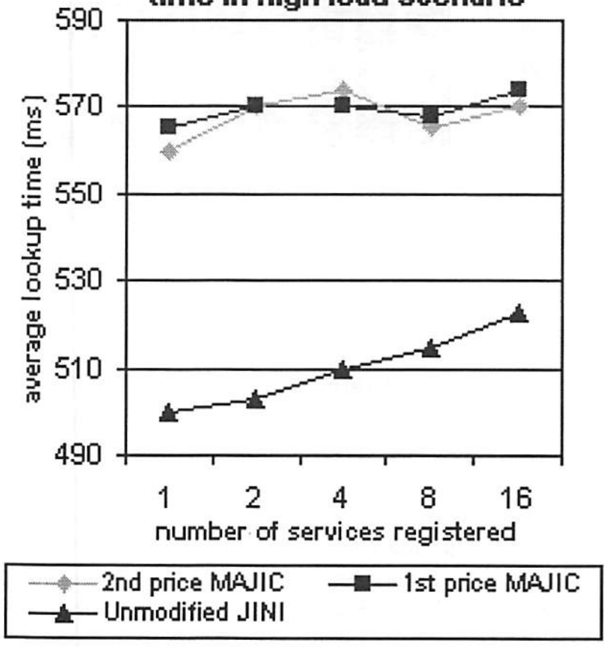
The flow of events of each test was as follows: (1) Activating a specific type of *lookup service*. (2) Activating the required service providers. (3) Activating the clients with configurable time interval between their invocations. (4) The clients initiate the *lookup* protocol and eventually invoke the given service.

5.2 Results

5.2.1 System performance

The performance of the system has been measured by the *lookup* protocol response time, since it is the most significant difference between the Jini and the MAJIC systems. This is measured at the buyer side and contains the lookup search time and the network latency. Fig 2 shows the performance results in high load scenario (no time interval between clients). In low load scenarios the results are similar, but the relative overhead is larger (see the full version of the paper [22]). The tests have been performed using 1000 clients and variable number of "simple" services. The main purpose of this test is to examine the MAJIC system overhead due to the market mechanism (including the parameter search engine) and the additional message (FC) that is being sent to the chosen seller after every assignment. We should emphasis that the additional overhead is only for the *lookup service* itself, which is normally insignificant compared to invocations of services. From these results, we see indeed that the MAJIC overhead is not prohibitive: in the low load case, we observed a 50% MAJIC overhead, while in the high load case we observed only 15% overhead. The difference between the two cases can be explained by the fact that in the high load scenario most of the *lookup* time is spent on waiting for entering the lookup service and therefore the MAJIC overhead is less significant.

Figure 2: Lookup response time in high load scenario



5.2.2 Load balancing

The load balancing tests have been performed on 8 printer services with 500 clients (500ms time interval between clients invocations). The services were identical in all parameters. Each seller's service-time was constantly updated to reflect its current load. Each buyer had a 60 ms job duration and a *utility* function: $u_B = 120 - 0.05 \cdot service - time$. Fig 3 shows the assignments of clients to services by the MAJIC system (for example, service number 1 was assigned to 64 clients). The average number of clients assigned to a service is 62.5, moreover, the maximal deviation from the average is 10%. Pure online algorithms geared to load balancing, which are 2-competitive, show similar results [12].

Figure 3: Load balancing in the MAJIC system

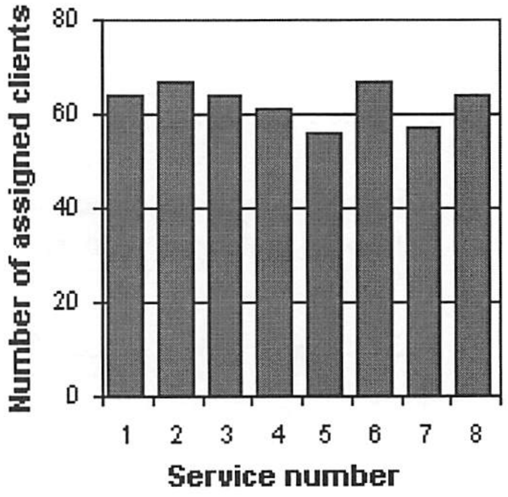
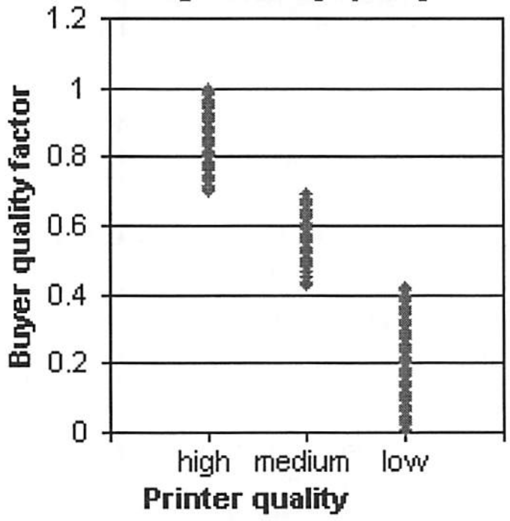


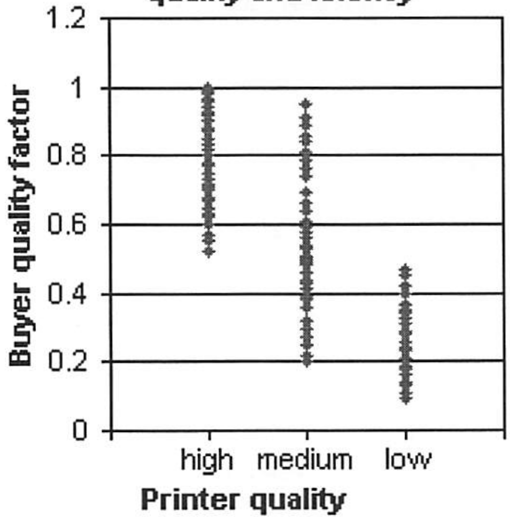
Fig 4: Assignment by quality



5.2.3 Resource allocation efficiency

In order to demonstrate resource allocation efficiency, we chose to perform tests on the *quality* attribute of a printer service. We have designed a system that contains printer services with different printing qualities. In this system, each buyer had a particular preference for a printing quality. We expected from the MAJIC system to assign buyers with preference for higher quality to high quality printers and vice versa. We have used 3 printer services with the following qualities: High (quality=150, price=15), Medium (quality=100, price=10) and Low (quality=50, price=5). Note that as quality decreases the printing price decreases as well. Each buyer has a parameter, f_q, which is a continuous quality factor that is chosen uniformly in the range of [0,1]. This factor represents the buyer's preference for printing quality. We activated 100 clients using the following utility function: $u_B = 40 + f_q \cdot quality$. As f_q increases, the buyer utility grows faster with quality. Thus, we expect that as the quality factor gets higher, the client will tend to be assigned to higher quality printer, although it charges higher price. In Fig 4, we show buyers assignments on the described services with respect to their quality . As we can see, buyer with higher quality factor is assigned to higher quality service. For example, the buyer with $f_q = 0.8$ is assigned to the High quality service.

Fig 5: Assignment by quality and latency



In addition, we have tested the same scenario when the printer service-time had also been taken into consideration: $u_B = 100 + f_q \cdot quality - 0.25 \cdot service\_time$. As we can see in Fig 5, the service-time parameter caused a load balancing effect; when one of the services was loaded, the clients that were supposed to be assigned to this service have been assigned to an adjacent service instead. We observe that even when the system manifests load balancing, the clients' quality preferences still effect the assignments.

6 Conclusions

We have introduced a blueprint for an infrastructure that performs on line auctions for computer services over distributed object systems. We implemented such a system on top of Sun's Jini system. We have presented both theoretical and initial empirical studies showing the efficiency of such systems. Two major aspects of our architecture distinguishes our work from previous related systems: (1) we provide a general-purpose architecture as opposed to previous economically-based systems that were dedicated to a single resources. (2) Our infrastructure handles a multiparameter space in a non trivial way.

The main future test that should be applied to our system is using it on a large scale for some specific service types. This way, we can examine some parameters of the MAJIC system: the system efficiency, possible implementations of *parameters search engines*, integration with existing security environments, etc.

Acknowledgement. We thank Ron Lavi, Ahuva Mu'alem and Zinovi Rabinovich for their helpful comments. We thank Yoav Etsion and Motti Beaton for technical assistance.

References

[1] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification.* Addison-Wesley Publishing, Reading MA, 1999.

[2] Y. Azar and L. Epstein. On-line load balancing of temporary tasks on identical machines. *5th Israeli Symposum on Theory of Computing and Systems*, pages 119 - 130, 1993.

[3] A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. van der Wijk, M. van Steen, and A. S. Tanenbaum. The globe distribution network. *Proc. 2000 USENIX Annual Conf.*, 1:141 - 152, 2000.

[4] E. H. Clarke. Multipart pricing of public goods. *Public Choice*, pages 17–33, 1971.

[5] *The OMG's CORBA website.* Web Page: http://www.corba.org/.

[6] *Microsoft DCOM technology: Distributed Component Object Model.* Web Page: http://www.microsoft.com/com/tech/DCOM.asp.

[7] W. K. Edwards. *Core Jini.* Prentice Hall PTR, Upper Saddle River NJ, 1999.

[8] R. El-Yaniv and A. Borodin. *On-Line Computation and Competitive Analysis.* Cambridge University Press, 1998.

[9] Carl A. Waldspurger et al. Spawn: A distributed computational economy. *IEEE Trans. on Software Engineering*, 18(2), 1992.

[10] D. F. Ferguson, C. Nikolaou, and Y. Yemini. Economic models for allocating resources in computer systems. In Scott Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation.* World Scientific, 1995.

[11] Michael R. G. and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of Np-Completeness.* W.H. Freeman and Co., 1979.

[12] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math*, 17:263 - 269, 1969.

[13] A. S. Grimshaw, W. A. Wulf, and the Legion team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), 1997.

[14] T. Groves. Incentives in teams. *Econometrica*, pages 617–631, 1973.

[15] B. Huberman. *The Ecology of Computation.* Elsevier Science Publishers/North-Holland, 1988.

[16] B. A. Huberman and T. Hogg. Distributed computation as an economic system. *Journal of Economic Perspectives*, pages 141–152, 1995.

[17] *Java home page.*
Web Page: http://java.sun.com.

[18] *Sun's JINI Homepage.*
Web Page: http://www.sun.com/jini.

[19] Y.A Korilis, A. A. Lazar, and A. Orda. Architecting noncooperative networks. *IEEE Journal on Selected Areas in Communication (Special Issue on Advances in the Fundamentals of Networking)*, 13(7):1241–1251, September 1991.

[20] A.A. Lazar and N. Semret. The progressive second price auction mechanism for network resource sharing. In *8th International Symposium on Dynamic Games*, Maastricht, The Netherlands, July 1998.

[21] J. K. Mackie-Mason and H. R. Varian. Pricing the internet. In B. Kahn and J. Keller, editors, *Public Access to the Internet.* Prentice Hall, 1994.

[22] *The MAJIC home page.* Web Page: http://www.cs.huji.ac.il/~liad/jini/index.html.

[23] A. Mas-Collel, W. Whinston, and J. Green. *Microeconomic Theory.* Oxford university press, 1995.

[24] P. Milgrom. Auctions and bidding: a primer. *Journal of economic perspectives*, 3(3):3 – 22, 1989.

[25] D. Monderer and M. Tennenholtz. Distributed games. To appear in Games and Economic Behaviour.

[26] N. Nisan. Algorithms for selfish agents. In *STACS*, 1999.

[27] N. Nisan, S. London, O. Regev, and O. Camiel. Globally distributed computation over the internet – the popcorn project. In *ICDCS*, 1998.

[28] N. Nisan and A. Ronen. Algorithmic mechanism design. In *STOC*, 1999. Avilable at http://www.cs.huji.ac.il/ amiry.

[29] N. Nisan and A. Ronen. Computationally feasible vcg-based mechanisms. In *EC*, 2000.

[30] M. J. Osborne and A. Rubistein. *A Course in Game Theory.* MIT press, 1994.

[31] Klemperer P. Auction theory: A guide to the literature. *Journal of Economic Surveys*, 13(3):227 – 286, 1999.

[32] C. H. Papadimitriou. Computational aspects of organization theory. In *Proceedings of the 1996 European Symposium on Algorithms.* Springer LNCS, 1996.

[33] O. Rees, N. Edwards, M. Madsen, M. Beasley, and A. McClenaghan. A web of distributed objects. *World Wide Web Journal*, 1(1):75 – 88, 1995.

[34] O. Regev and N. Nisan. The popcorn market - an online market for computational resources. In *ICE*, 1998.

[35] *Java's Remote Method Invocation.* Web Page: http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[36] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers.* MIT Press, 1994.

[37] Shenker S. Making greed work in networks: A game-theoretic analysis of switch service disciplines. In *Proc. of the ACM SIGCOMM*, 1994.

[38] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI-99*, 1999.

[39] T. W. Sandholm. Limitations of the vickrey auction in computational multiagent systems. In *Proceedings of the Second International Conference on Multiagent Systems (ICMAS-96)*, pages 299–306, Keihanna Plaza, Kyoto, Japan, December 1996.

[40] D. Estrin D. Shenker, S. Clark and S. Hertzog. Pricing in computer networks: Reshaping the research agenda. *ACM Computational Comm. Review*, pages 19–43, 1996.

[41] *Microsoft SOAP: The Simple Object Access Protocol.* Web Page: http://www.microsoft.com/mind/0100/soap/soap.asp.

[42] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, and C. Yoshikawa. Webos: Operating system services for wide area applications. *The Seventh IEEE Symposium on High Performance Distributed Computing*, 1998.

[43] H. R. Varian. Economic mechanism design for computerized agents. In *Proceedings of the First Usenix Conference on Electronic Commerce*, New York, July 1995.

[44] W. Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance*, pages 8–37, 1961.

[45] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational ecology. *IEEE Transactions on Software Engineering*, 18(2), 1992.

[46] W.E. Walsh and M.P. Wellman. A market protocol for decentralized task allocation: Extended version. In *The Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, 1998.

[47] W.E. Walsh, M.P. Wellman, P.R. Wurman, and J.K. MacKie-Mason. Auction protocols for decentralized scheduling. In *Proceedings of The Eighteenth International Conference on Distributed Computing Systems (ICDCS-98)*, Amsterdam, The Netherlands, 1998.

[48] *WebBroker: Distributed Object Communication on the Web.* Web Page: http://www.w3.org/TR/1998/NOTE-webbroker/.

[49] *XML home page.* Web Page: http://www.xml.com.

End-to-end WAN Service Availability*

Bharat Chandra, Mike Dahlin, Lei Gao, and Amol Nayate
University of Texas at Austin

Abstract

This paper seeks to understand how network failures affect the availability of service delivery across wide area networks and to evaluate classes of techniques for improving end-to-end service availability. Using several large-scale connectivity traces, we develop a model of failures that includes key parameters such as failure location and failure duration. We then use trace-based simulation to evaluate several classes of techniques for coping with network failures. We find that caching alone is seldom effective at insulating services from failures but that the combination of mobile extension code and prefetching can improve failure rates by as much as an order of magnitude for classes of service whose semantics support disconnected operation. We find that routing-based techniques may provide significant improvements, but that the improvements of many individual techniques are limited because they do not address all significant categories of network failures. By combining the techniques we examine, some systems may be able to improve availability by as much as one or two orders of magnitude.

1 Introduction

This paper seeks to understand how network failures affect the availability of service delivery across wide area networks (WANs) and to evaluate classes of techniques for improving end-to-end service availability. By providing a quantitative analysis of these techniques, we hope to provide a framework to help service designers select from and make best use of currently-available techniques. Further, we seek to evaluate the potential impact on availability from proposed extensions to the Internet infrastructure such as replication of active objects [1, 3, 13, 29, 30, 35] and overlay routing [26].

*This work was supported in part by an NSF CISE grant (CDA-9624082), the Texas Advanced Technology Program, the Texas Advanced Research Program, and grants from IBM, Novell, and Tivoli. Dahlin was supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Research Fellowship.

Although several commercial hosting services today advertise 99.99% or 99.999% ("four 9's" or "five 9's") server availability, providing highly available servers is not sufficient for providing a highly available service because it is not an end-to-end approach: other types of failures can prevent users from accessing services. Internet connectivity failures, unfortunately, are not rare. Paxson [23], for example finds that "significant routing pathologies" prevent selected pairs of hosts from communicating about 1.5% to 3.3% of the time, and more recent measurements [36] suggest that availability has not significantly improved since then. In contrast with the 5 minutes per year of unavailability for a five-9's system, a typical two-9's Internet-delivered service will be unavailable for nearly 15 minutes per day from a typical client.

Although caching can improve file system availability [12, 14], there is reason to be concerned that caching alone will not significantly improve WAN service availability because much HTTP traffic is uncachable [7, 34]. This limitation motivates us to study the potential effectiveness of other techniques such as hoarding [14], push-based content distribution [9], relaxed consistency, mobile extensions to ship service code to proxies or clients [1, 3, 13, 29, 30, 35], anycast [2, 8, 35], and overlay routing [26]. Although the performance benefits of many of these techniques have been studied, their potential impact on end-to-end availability has not been quantified.

Our analysis faces two challenges. First, we wish to evaluate the potential effectiveness of a wide range of techniques for a wide range of services. To do this, we abstract away both the detailed design of the techniques and the semantic requirements of the services. By using these simplifications, we can determine upper bounds on improvements that different classes of techniques can yield. To refine these simple bounds, we then explore the sensitivity of the techniques to parameters representing factors that could limit their effectiveness. The second challenge is that available studies of WAN failure patterns do not quantify several important parameters. To address this, we analyze connectivity traces to develop a model suitable for evaluating techniques for coping with failures.

This work makes three contributions. First, we develop a WAN connectivity failure model that includes failure rate, failure location, and failure duration. A key finding is that failure duration distributions appear heavy-tailed, which means that long failures account for a significant fraction of failure durations. Second, we conclude that data-caching-based techniques for improving service availability will likely have little success, but that the combination of prefetching and shipping mobile extension code to clients and proxies has the potential to improve failure rates by over an order of magnitude. Unfortunately, three factors may significantly limit these gains: (i) compulsory misses to extension code and state, (ii) capacity misses due to limitations in the number of extensions a client or proxy can host, and (iii) service-specific semantic requirements that prevent some services from using these techniques. Finally, we find that routing-based approaches can significantly improve failure rates, but that near-client, near-server, and interior network failures all contribute significantly to the overall failure rates, which limits end-to-end improvements from efforts that address only one type of problem (e.g., multi-hosting a server with multiple ISPs).

The rest of this paper proceeds as follows. We first discuss related work in the areas of coping with network failures and modeling Internet failure patterns. We then describe the network failure model we have developed. Section 4 evaluates classes of techniques for coping with network failures when delivering Internet services. Finally, Section 5 summarizes our conclusions.

2 Related work

The basic techniques we examine for improving robustness have been studied in other contexts. In file systems, caching, hoarding, and relaxed consistency can isolate clients from network and server failures [12, 14, 27]. Odyssey [21] explores using application-specific adaptation to cope with disconnection by dynamically adjusting service semantics.

In the context of web services, previous studies have examined the performance benefits of caching [7, 28, 33], prefetching [6, 22, 16, 24], pushing updates [18, 28], push-based content distribution [9], server replication [20], mobile code [1, 3, 13, 29, 30], and overlay routing [26], but the impact on end-to-end service availability of these techniques has not been systematically quantified.

Systems implementing variations of some of these techniques have been built. The Netscape Navigator browser supports "off-line" browsing from its cache and and the Microsoft Internet Explorer Browser supports hoarding. Joseph et al.'s Rover toolkit [13] is designed to support disconnected operation for mobile clients accessing services. But these techniques have not been systematically applied to or evaluated for large numbers of services.

Paxson studies IP-level routing pathologies and finds that "major routing pathologies" thwart IP routing between a given pair of hosts 1.5% to 3.4% of the time [23]. The study focuses on quantifying the prevalence and diagnosing the causes of IP-level failures. Our analysis builds on this study by studying these traces to determine metrics relevant to end-to-end service delivery: failure location and duration.

Labovitz et al. [17] examine route availability by studying routing table update logs. They find that only 25% to 35% of routes had availability higher than 99.99% and that 10% of routes were available less than 95% of the time. They find that 60% of failures are repaired in a half hour or less, and that the remaining failures exhibit a heavy-tailed distribution. These results are qualitatively consistent with our end-to-end analysis and provide additional evidence that connectivity failures may significantly reduce WAN service availability.

Zhang et al. [36] study NIMI and traceroute measurements taken during December 1999 and January 2000. They find that routing reliability has neither degraded nor improved significantly since Paxson's 1995 study. The focus of this study is on stationarity of network behavior, and it finds considerable variation in behavior at different network locations, at different times, and on different time scales. This points to a potential limitation of our current study, which uses average behavior across paths and over time to develop a model of average availability. Given the variability found by Zhang et al. for the metrics of route stability, packet loss, and throughput, future work to study the impact of route variability of connectivity would be valuable.

3 Network failure model

We seek to model parameters of network failures that most directly affect techniques to improve availability. The most basic parameter is failure rate: what fraction of time are two nodes unable to communicate? We then analyze failure patterns along two dimensions: failure location and failure duration.

Failure duration is important because it influences the effectiveness of techniques for coping with failures. For example, it may be simpler to use caching or prefetching to mask short failures than long failures since masking long failures requires predicting access patterns across longer periods of time, transferring more data to the cache, and storing more data in the cache.

Failure location is important because it influences the effectiveness of routing-based strategies. We use a simple model that classifies failures into three operationally significant categories – "near-source," "in-middle," and "near-destination." Near-source failures represent failures of the client stub network that disconnect a source machine or source subnet from the rest of the Internet. Near-destination failures have a similar effect on destinations. In-middle failures represent connectivity failures in the middle of the network that prevent a pair of nodes from contacting one another (on the default route), but where both nodes are able to contact a significant fraction of the remaining nodes on the Internet. We also use the term "stub failures" to refer to the combined near-source and near-destination categories.

This location model is admittedly simplistic. Most notably, it represents in-middle failures as the interruption of connectivity between a single pair of nodes that does not affect any other pairs' ability to communicate. In reality failures in the middle of the Internet infrastructure will typically affect more than one pair of nodes [17]. Thus, groups of such middle failures are likely to be correlated, and the correlation will depend on details of network topology that would be complex to model. However, we believe that our simple model provides a reasonable first-order approximation for evaluating routing based techniques: an in-middle failure represents the case where both the source and destination can connect to a nontrivial fraction of the Internet but cannot connect to each other by the default route. Assuming that the core Internet is not partitioned, routing-based techniques are likely to be able to find an alternate route between the nodes in such a situation.

Our failure model uses a simple model for inter-arrival times. Given a failure rate (expressed as a fraction of time a particular class of failures occurs at a particular location) and an average failure duration, we calculate the average inter-arrival time for each class of failures. We then assume that failures arrive independently with exponentially distributed inter-arrival times with the given average arrival rate. We leave as future work extending the model to (a) model correlation in time of arrivals to a location, (b) model correlation of arrivals across different locations, (c) model time-of-day dependencies, and (d) model different failure patterns on different paths [36].

Parameter	Default value	Comment
Rate	1.5% (all failures) 1.25% (> 30s)	Varies from .4% to 7.4% in different data sets
Location	Src: 25% Mid: 50% Dest: 25%	All locations significant. Ratio varies widely across traces.
Duration	avg = 609 sec. $\text{pdf}(x) = 16x^{-1.85}$	Appears heavy-tailed
Interarrival	avg = 48111 sec.	

Table 1: Default parameters for failure model.

Another enhancement to the model left as future work is modeling quality of service. Whereas our simple model tracks periods of complete disconnection, for some applications, the network has "failed" if the bandwidth falls below a certain level or the latency rises above some level. A more sophisticated failure model might account for variations in quality of service as well as the coarse metric of connectivity on which we focus.

Table 1 summarizes key parameters for our model. The following subsections describe how these parameters are obtained.

3.1 Failure analysis methodology

Our basic methodology for quantifying failure patterns uses datasets that consist of large numbers of attempts by pairs of nodes to communicate. We identify attempts that succeed and those that fail and use this information to develop models for the frequency, duration, and location of failures in the Internet.

We use two types of dataset. First, traceroute datasets consist of multiple traceroute measurements between pairs of nodes participating in the study. Second, HTTP datasets consist of logs of HTTP requests through public Squid [31] proxies to web servers. The datasets used are described in more detail in the next subsection.

There are potential biases in our approach resulting from both limitations of our data sets and our analysis of them.

First, the hosts and network paths that we trace may not be representative of typical Internet connectivity. Several of our traceroute datasets were collected by Paxson, and he argues that the interior nodes measured may be representative of typical routes but that the end-hosts may not be [23]. Other traceroute datasets were gathered by Savage et. al [26] from sites selected for convenience. Although our HTTP traces are sent to a collection of servers dominated by publicly-available HTTP servers, requests are sent from regional Squid proxies. These Squid proxies may be unusual sources both in terms of their network connectivity and in terms of the user community they serve.

Although we seek to develop end-to-end failure models, our data sets are not, strictly speaking, end-to-end. In particular, the traceroute data sets track failures at the IP level but omit higher-level protocol failures such as DNS failures. Also, because traceroute server machines' failure patterns may not be representative of those of HTTP server machines, we filter out "end-host" failures from the traceroute data sets. These factors mean that we may underestimate end-to-end failure rates.

Also, our data sets may under-report the number of failures that happen near the source node of a request. In both the HTTP and traceroute data sets, network disruptions near the intended source of a measurement may prevent requests from being issued during these periods when they are more likely than average to fail.

Another source of bias is the data sampling patterns used in some data sets. The traceroute data sets (except uw-1) use exponentially-distributed random inter-measurement times. By the PASTA (Poisson Arrivals See Time Average) principle [32], the fraction of requests that fail in the traceroute experiment should correspond to the fraction of time the network is down (neglecting the source failure sampling bias listed above). The HTTP data sets sample routes according to the request pattern from clients and therefore the samples reflect the request-average behavior of the system, but this may differ from the time-average behavior of the system because the state of the network may affect whether a trace sample is taken or not. For example, if a user's first request to a server fails, it is unlikely the user will send additional requests to the server in the near future.

3.2 Datasets

Table 2 summarizes the traces we use to construct our network failure model.

Paxson-1 and Paxson-2 are traceroute measurements taken and originally analyzed by Paxson [23]. In Paxson-1 each site sends a probe to a randomly chosen target with an exponential inter-probe interval of 2 hours. The number of sites varies over the course of the trace up to a maximum of 27 nodes. In Paxson-2, 40% of measurements from a site are to a randomly chosen target site with exponential inter-probe intervals of 2 hours. The remaining 60% of a sites measurements are sent in "bursts" with the same 2-hour inter-probe interval but without changing the target from the previous probe.

Traceroute Datasets

Dataset	Year	Duration	nhosts	nsamples
Paxson-1	1994	45 days	27	7016
Paxson-1-na	1994	45 days	22	4903
Paxson-2	1995	48 days	33	28943
Paxson-2-na	1995	48 days	23	12613
uw-1	1999	34 days	36	54391
uw-3	1999	7 days	36	78816
uw-4a	1999	14 days	14	181151
uw-4b-all	1999	12 days	38	58488

HTTP Datasets

Dataset	Year	Duration	nhosts	nsamples
Bol	2000	16 days	1/194284	8142820
Rtp	2000	12 days	1/282830	18577435
Squid2	2000	3 days	9/327835	23490956

Table 2: Network failure traces. For traceroute traces, nhosts is the number of participating nodes; each node acted as both a source and a destination. For HTTP traces, nhosts shows {the number of proxy caches traced}/{the number of servers they contacted.} Nsamples shows the number of attempts to communicate in each trace.

Paxson-1-na and Paxson-2-na represent the subset of measurements in the Paxson traces that both begin and end in North America.

uw-1, uw-3, uw-4a, and uw-4b-all are traceroute traces collected by Savage et. al [26] at the University of Washington. In uw-1, the inter-measurement time is a uniform distribution with a mean of 15 minutes and each measurement is between a random pair of hosts. In uw-3 and uw-4b-all a random pair of hosts is selected for each measurement using an exponential distribution with a mean of 9 and 150 seconds, respectively. In uw-4a, every server sends requests to every other server at the same time; these episodes are scheduled using an exponential distribution with mean of 1000 seconds.

A problem with uw4a is self-interference. Approximately 10 requests are issued by each node "simultaneously", which may increase packet losses. To reduce this effect, we filter obvious cases of self interference: if at least one outbound packet in a burst of requests from a node makes it to its destination, then we conclude that connectivity from that node to the Internet is available at the time of the burst. If any other traceroute during the burst fails to make it beyond the source node subnet or the "bottleneck" routers that are traversed on all successful outbound requests from that node, we conclude that traceroute was a victim of self-interference and discard it from the trace set. We use a similar procedure to filter bursts of inbound traceroutes to destinations. Overall, we delete 1.6% of the requests from uw4a due to self-interference.

Bol, Rtp, and Squid2 are traces of HTTP re-

	Temp	Perst	Total
Paxson1	1.3%	0.43%	1.7%
Paxson1-na	1.4%	0.48%	1.9%
Paxson2	1.7%	0.19%	1.9%
Paxson2-na	0.60%	0.072%	0.7%
uw1	NA	0.15%	NA
uw3	NA	0.027%	NA
uw4a	NA	0.61%	NA
uw4b-all	NA	0.0047%	NA
Bol			7.4%
Rtp			1.5%
Squid2			1.1%

Table 3: Fraction of requests that fail.

quests taken at proxy caches that are part of the Squid cache hierarchy [31]. Bol and Rtp are from individual proxies, and Squid2 combines requests from nine proxies. We first filter the trace to remove the 22.6% of requests satisfied locally (e.g., a cache hit) or indirectly (e.g., via a sibling cache). We then filter all TCP_REFRESH_MISS requests from the trace because such requests fail a disproportionate fraction of the time (80% to 90% of the TCP_REFRESH_MISS requests fail in most of the traces.) We ignore requests with reply code 400 or 500 (which account for 0.37% of all replies) because it is ambigous whether connections were successful in these cases. We then count requests with code 504 ("Gateway time out") as failed connections, and we count the remaining requests as successful network connections from the proxy to the server.

3.3 Failure rates

For the Paxson data sets, we find "temporary" failure rates (where connectivity is interrupted for at least 30 seconds during a traceroute episode but where the traceroute episode eventually succeeds in contacting its target) of 0.6% to 1.7%, and find "persistent" failure rates (where the traceroute fails to reach its destination) of 0.07% to 0.48% when end-host failures are excluded. The overall failure rates of these traces is 0.7% to 1.9%.

We find similar failure rates for the uw and HTTP traces. As Table 3 shows, the uw data sets show similar persistent failure rates to the Paxson traces (though the temporary failures are, unfortunately, not included in the uw data sets.) The Rtp and Squid2 traces' failure rates are similar to the overall traceroute rates – 1.5% and 1.1%. The Bol trace shows a higher rate, and we note that the component traces of the Squid2 data set show considerable variability, with individual proxies exhibiting failure rates of 0.37%, 0.5%, 0.67%, 0.85%, 1.2%, 1.6%, 1.8%, 3.6%, and 6.8%.

Overall, the data suggest that typically 0.5% to 2% of requests fail to communicate with their server, but some proxies differ considerably from this typical behavior and see rates as low as 0.36% or as high as 7% in our data sets. Our simulations in Section 4 will use a default failure rate of 1.25%. This means that a given pair of nodes is unable to communicate 1.25% of the time due to network failures lasting 30 seconds or longer.

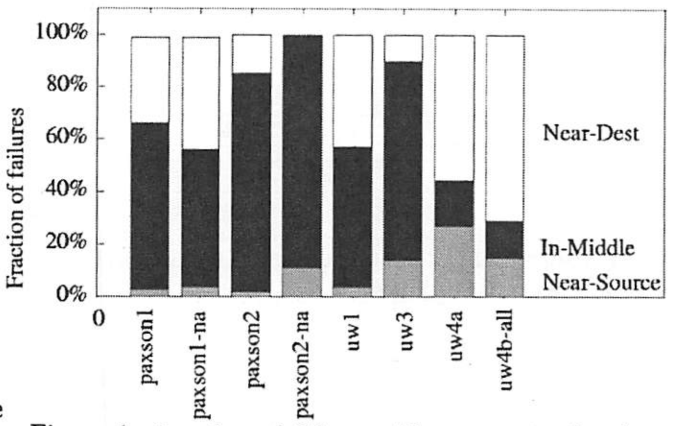


Figure 1: Location of failures. The segments of each bar show the fractions of failures that occur at particular locations.

3.4 Failure locations

We focus on the traceroute data sets because they include hop-by-hop routing information, and we use the following heuristics to classify failures into the near-source, in-middle, and near-destination categories. We define the *source bottleneck set* as the set of routers that are visited by all successful outgoing requests from a node and the *source subnet set* as the set of routers whose IP addresses match the source node's in the top 24 bits. We define the destination bottleneck and subnet sets similarly. A failed request is classified as a near-source failure if (a) the request only succeeds in reaching nodes in the source bottleneck set or source subnet set or (b) two or more successive requests from the same source to different destinations fail. We use a similar definition for the near-destination failures. We classify all remaining failures as in-middle failures.

Figure 1 summarizes the fraction of failures classified as near-source, near-destination, or in-middle by these criteria.

As noted earlier, the methodology used for gathering the traces may tend to undersample during periods when the network near the intended source of the measurement is malfunctioning. As one might therefore expect, the near-destination failure rate is higher than the near-source failure rate in most of the

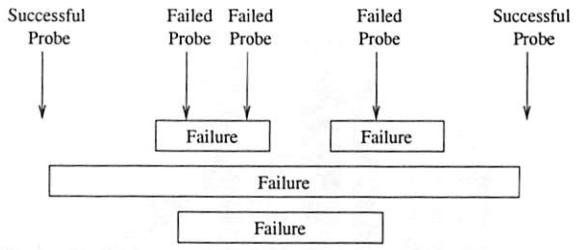


Figure 2: Illustration of ambiguity in failure event duration from probe samples.

data sets. Given that for these data sets the source and destination nodes were selected from the same collection of traceroute hosts, we speculate that the near-destination failure rates reported above are more representative of the stub network disconnection rate than the near-source rates.

Overall, we observe that both stub network and interior failures contribute significantly to failures but that the relative prevalence of interior compared to stub failures varies. Paxson2, Paxson2-na, and uw3 are dominated by interior failures, Paxson1, Paxson1-na, and uw1 have similar amounts of interior compared to stub failures, and the other traces are dominated by stub failures. Our simulation by default categorizes failures as near-source, in-middle, and near-destination in a ratio of 1:2:1.

3.5 Failure durations

As Figure 2 illustrates, the relatively low sampling rate at some locations and data sets can lead to two ambiguities for estimating the duration of a failure event. First, the samples shown could either be from one long failure or two (or more) short failures. Second, the beginning of the failure could have occurred soon before the first probe that failed or soon after the last probe that succeeded; there is a similar ambiguity for the ending time. For our baseline model, we assume that any series of failures without an intervening success represents a single failure event, and we use the data to provide both upper and lower bounds on the duration of each such event. An area for future work is developing failure data sets that provide better failure-duration information.

Figure 3 shows the cumulative distribution function of the duration of failure events lasting longer than 30 seconds for the http data sets. Our rationale for only looking 30 seconds or longer failures is that short failures may be better handled by transport-level retransmission than the more aggressive techniques we explore. Excluded sub-30-second failures account for 28.8% of the failed probes and 70.7% of the failure events for the lower bound list of durations; they account for 6.5% of the probes and 31.7% of the events for the upper bound list.

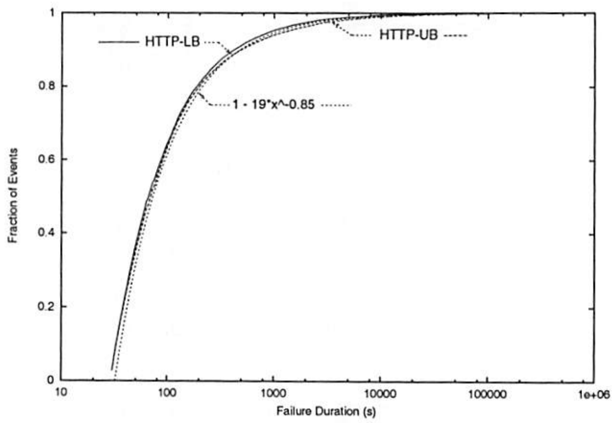


Figure 3: Cumulative distribution function of failures lasting longer than 30 seconds in the combined HTTP data sets.

This distribution appears well modeled by the function $F(x) = 1 - 19x^{-0.85}$. The average duration of this function is unbounded. For our simulations, we arbitrarily place an upper limit on failure durations of 500,000 seconds, which yields an average failure duration of 609 seconds.

For the traceroute data sets, the long inter-probe times make it difficult to precisely characterize the duration of short persistent failures. Figure 4 compares the duration of long failure events – 1000 seconds or more – between the traceroute and HTTP data sets. For clarity, we combine all of the traceroute failures into one data set and all of the HTTP failures into another one and show the upper and lower bounds of the cumulative distribution function for each. Note that the traceroute lower bound line is to the right of the upper bound line for much of the range because the two lines track different sets of data once the sub-1000-second events are excluded.

Based on the data in Figures 3 and 4, it appears that the duration of 30+-second HTTP and 1000+-second traceroute failure events display cumulative distribution functions of the form $F(x) = 1 - (k/x)^{\alpha}$ with $\alpha \approx 0.85$. This function corresponds to a *heavy-tailed* distribution typified by a significant number of long failures, decreasing recovery rate, large variance, and high mean [10].

The traceroute and HTTP data sets each have significant limitations for the purposes of modeling failure duration: the HTTP data sets may contain sampling-interval biases, and the traceroute data sets' sparse sampling interval leaves uncertainty about the duration of individual events. Despite these limitations, the data sets appear qualitatively consistent with one another, which suggests that the results are not anomalous.

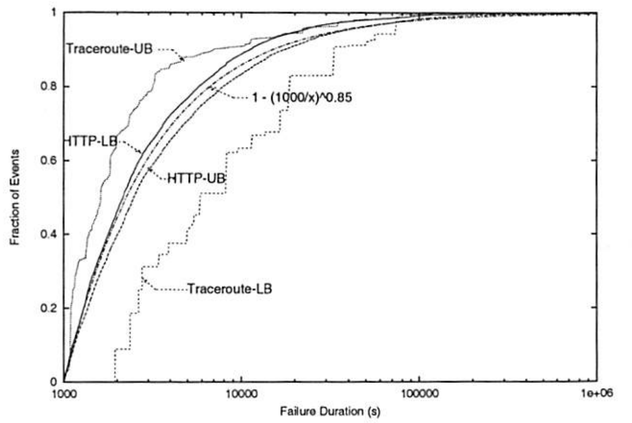


Figure 4: Cumulative probability function of the duration of failures lasting longer than 1000 seconds. The x axis is the duration and the y axis is the probability that a failure event will have less than the specified duration.

4 Masking network failures

This section studies two classes of techniques for improving end-to-end service availability by masking network failures. Client-independence techniques – such as data caching, prefetching, and mobile code – provide a (possibly degraded) version of a service using local resources when the remote server cannot be contacted. Routing and connectivity techniques use alternate network paths to route around failures.

These experiments focus on two goals. First, they seek to quantify the potential effectiveness of these techniques at improving service availability. In order to provide information about of a broad range of techniques, our experiments abstract away implementation details and thus provide an upper bound on the techniques' effectiveness. The second goal of our experiments, therefore, is to understand what factors may limit specific instantiations of these techniques and to quantify their impact.

Although this paper focuses on service-level techniques for improving availability, researchers will certainly work to improve reliability at the hardware and transport layers as well. Indeed, achieving the goal of four- or five-nine services will likely require advances at all layers. So, in addition to the experiments described above, we assess the sensitivity of our results to changes in the reliability of the underlying infrastructure.

4.1 Client independence

A range of client independence techniques are available.

1. **Caching.** Caching hides network and server failures by serving requests from a nearby cache rather than a distant server [12]. Most web clients today include some form of caching.

2. **Relaxed consistency and push-updates.** Relaxed consistency can improve availability by allowing caches to serve potentially stale data during failures rather than requiring the cache to use (unavailable) current data. Alternately, under a push-updates protocol [18, 28], servers may update cached copies before clients issue reads requesting the new versions. Push-updates thus improves the chance that a cache will contain current data during a disconnection.

3. **Prefetching.** Prefetching brings objects close to a client before the client accesses them. **Hoarding**, a form of prefetching in which a user identifies groups of objects to fetch, is effective for disconnected operation in file systems [14], and the Microsoft Internet Explorer browser implements a hoarding option for web pages. **Server push** [9] such as the content distribution networks becoming commercially available can be thought of as a form of server-directed prefetching. Note that prefetching is more aggressive than the "push update" approach described in the previous paragraph. "Push update" only distributes new versions of objects that have already been referenced by a cache, while prefetching can distribute unreferenced objects in order to avoid compulsory misses.

4. **Replication of active objects.** Several researchers have proposed systems in which active service objects may be cached or replicated and then executed [1, 3, 13, 29, 30]. These techniques may provide ways to extend the benefits of caching, relaxed consistency (or "application-specific adaptation" [21]), and prefetching to the significant fraction of web services that are not cachable [7, 34].

This set of experiments examines the potential effectiveness of using these client independence techniques to improve robustness of Internet services by transforming *failed sessions* that are interrupted by network disconnections into *degraded sessions* that are served by the cache or by downloaded mobile extensions. Clearly, the relative advantage of degraded sessions over failed sessions will vary from service to service: some services can provide full service while disconnected, others can provide tolerable service across short disconnections, and still others require continuous on-line communication with a remote site to be effective. To cope with this wide

Workload	Date	Clients	Servers	Sessions
Squid-P	3/28/00–4/03/00	1	131193	1557875
Squid-C	3/28/00	107	52526	403235
BU-P	1/17/95–5/17/95	1	4614	56789
BU-C	1/17/95–5/17/95	33	4614	68949

Table 4: Web access trace parameters.

range of service behaviors, this experiment does not attempt to quantify the benefit of degraded service over failed service; instead it seeks to quantify how often services have the option to use caching, relaxed consistency, prefetching, or mobile extensions to improve their robustness to network disconnections.

Workload and methodology. In addition to the failure model described above, the simulator uses two sets of web service access traces to represent Internet service access patterns. Table 4 summarizes key parameters for these traces. We examine both the Bol Squid trace described earlier and a four-month trace taken at clients at Boston University [5]. This trace is old, but it includes client cache hits, and the client-ID mappings are not changed over the trace period. We examine both traces from the point of view of a proxy shared by all clients in the trace (Squid-P and BU-P) and from the point of view of individual client machines (Squid-C and BU-C) with no shared proxy. Because the Squid traces change the client-ID mappings daily, we only look at the first day of the Squid-C trace.

For our simulations, we post-process the traces to group individual accesses into *sessions.* We define a session as a set of accesses from a client (-C traces) or proxy (-P traces) to a single server in which the maximum gap between successive requests is 60 seconds. Our figure of merit for availability is the fraction of sessions that complete without interruption.

Our simulator tracks the references to objects in the traces and uses trace information to classify the objects as cachable or uncachable and to identify when objects change. It assumes that each simulated client (-C traces) or proxy (-P traces) has an infinite cache that stores all objects accessed previously in the simulation.

To evaluate prefetching techniques and mobile code as a class without knowing the details of each service, we use the simulation parameter *install_time* to represent the amount of time from the first access by a client or proxy to a service until the service has downloaded sufficient state or programs or both to the cache to cope with network disconnections. Our default *install_time* is 100 seconds. During the *install_time*, clients and proxies must access the service from cached data or via the origin server.

If the network remains up during an entire session, the simulator classifies the session as *No Failure.* For sessions in which the network fails, the simulator examines the objects referenced in the session and classifies the session as follows: *Cache Hit* if all requests are for fresh cached web objects; *Stale Hit* if all requests are for cached web objects and if some of those objects require updates from the server; *Hoardable Degraded* if the *install_time* for the service has completed at time of the failure and all requests are for cachable objects but some miss; *Dynamic Degraded* if the *install_time* has completed at the time of the failure but not all session data are cachable; and *Fail* if the *install_time* has not completed at the time of the failure and either some data are not cachable or some data are cache misses.

For these experiments, we set the failure-location distribution to make all failures "in-middle" failures, and we conduct five trials with different random seeds for the network failure model and graph the mean and standard deviation of results. We describe improvements to failure rates in terms analogous to the common definition of "speedup" [11]: $improvement = \frac{failureRate_{orig}}{failureRate_{new}}$.

Results. First, we examine the effectiveness of these general techniques as well as the extent that installation time limits improvements. The y-axis of Figure 5 shows the fraction of sessions classified in the categories listed above on a logarithmic scale so that equal intervals reflect equal improvements to failure rates. The x-axis shows the *install_time* for each service also using a log scale, and each graph shows these results for a different workload. When installation times are short, the combined effect of all techniques is to improve the failure rate by at most factors of 14.4 (Squid-P), 15.4 (BU-P), 2.7 (Squid-C) and 5.22 (BU-C) for the four workloads compared to the failure rate that would be encountered if each request were sent to the origin server.

The improvements available from caching alone appear small (improvements to failure rates of 1.1, 1.6, 1.1, and 1.4 for caching and of 1.1, 1.6, 1.1, and 1.4 for caching plus relaxed consistency or push-updates). Note that the Squid workload's lower-level caches may hide sessions that only reference cached data, causing us to understate the benefits of caching alone. Conversely, the BU trace are not filtered by caches, but they are old and may reflect a workload that is unrealistically easy to cache. It seems likely that caching's benefits lie between these values.

In contrast with caching alone, aggressive prefetching plus caching may be able to achieve signif-

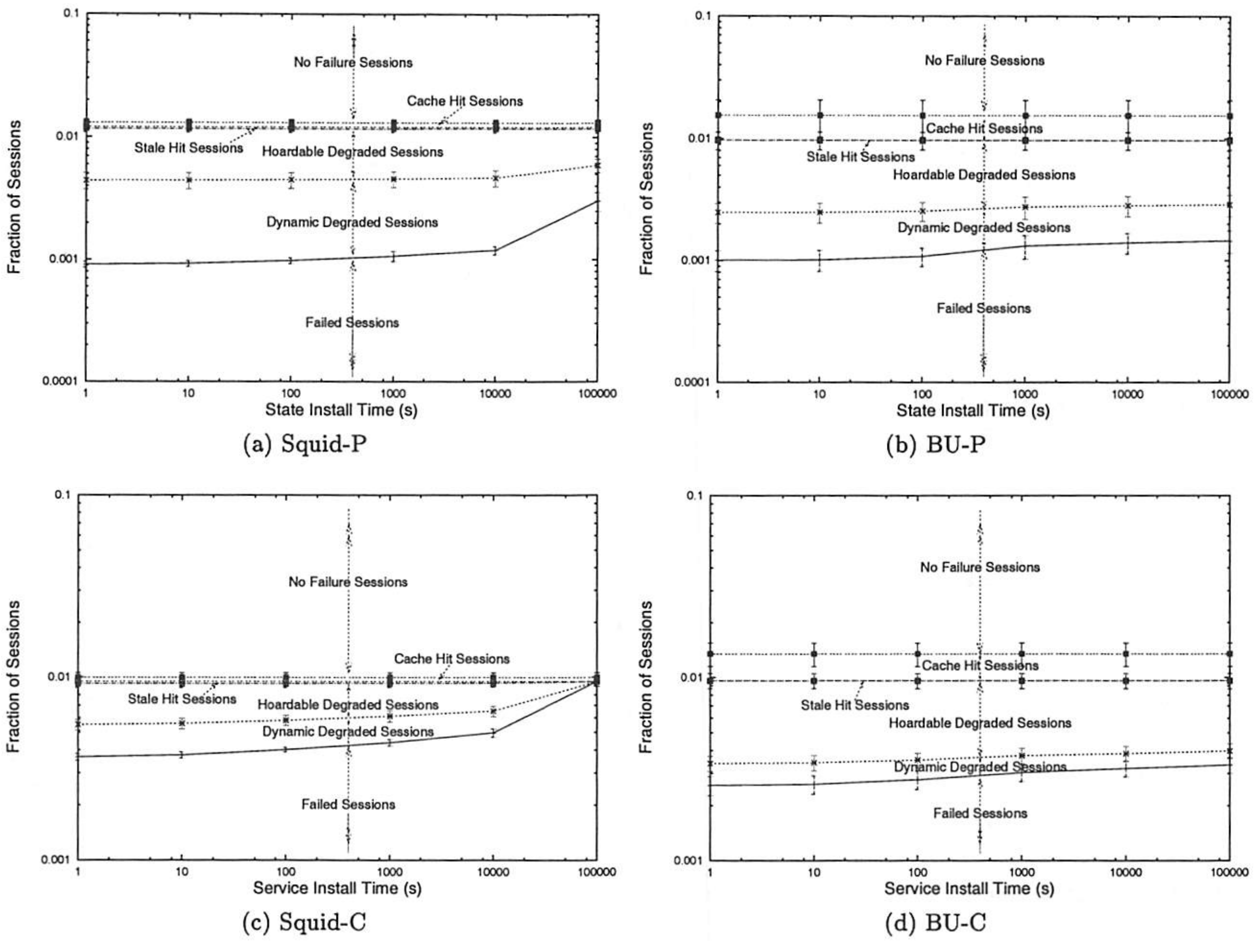


Figure 5: Session result v. state installation time. Each region between two lines represents the fraction of sessions that can be handled by the specified technique plus those above it in the graph.

icant improvements for those services where prefetching is feasible; the simulations indicate upper bounds of 3.0, 6.2, 1.8, and 4.0 for this combination.

The only limiting factor to active object replication in this model is our assumption that each service requires different extension code and data, and that extensions cannot be downloaded until a service is first accessed. Under this assumption, improvements to failure rates are limited to about an order of magnitude for these traces because if the network is down when a service is first accessed or during the first *install_time* of accesses, no code and data is available to mask the failure. These "compulsory misses" also limit the prefetching line in these graphs. If compulsory misses and initialization times are ignored, prefetching could provide improvements in failure rates of up to 3.7, 9.7, 4.7, and 12.2 and replication of active objects and their data could, in principal, provide at least degraded service 100% of the time.

The available benefits fall gradually as installation time increases and compulsory misses become more expensive. At a 10,000 second installation time the upper bound on availability improvements are 11.0, 11.1, 2.0, and 4.2 for the four workloads. This result is promising: it suggests that services that need to download significant amounts of state to provide acceptable disconnected service may have the opportunity to do so.

Next, we examine the sensitivity of our results to the underlying network failure rate. Figure 6 shows session results for Squid-P as we vary network failure rates by reducing the time between failures and leaving the failure duration distribution unchanged. The other workloads (not shown) are qualitatively similar. These data suggest the improvement in session failure rates provided by caching, prefetching, and replicas of active objects are relatively insensitive to the underlying network failure patterns between failure rates of .0125% and 12.5%.

The experiments above suggest that to significantly improve overall service availability, services

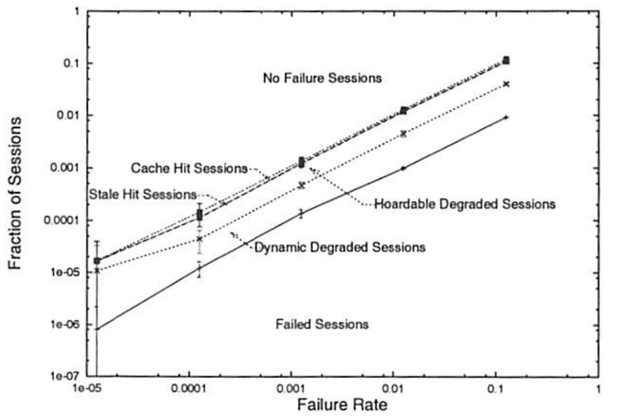


Figure 6: Session results (Squid-P) as network failure rates vary.

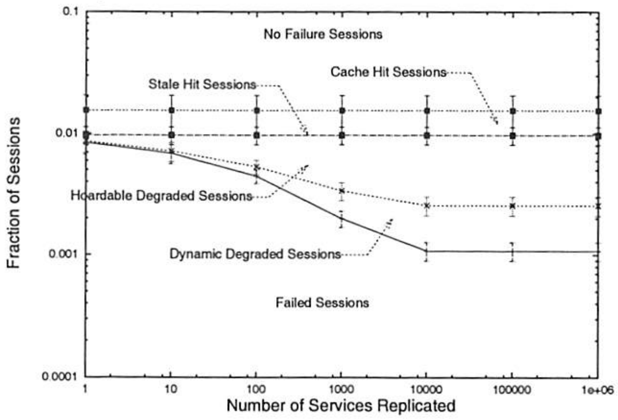


Figure 7: Session failure rate v. number of cached service extensions (BU-P trace).

may need to resort to prefetching and mobile extensions rather than relying on caching alone. Unfortunately, these techniques can dramatically increase the demand for resources at a client, proxy, or network. A key limiting factor, therefore, may be how many resources a cache can devote to each hosted service and how many services a cache can simultaneously host. Figure 7 shows session results when the BU-P proxy maintains only a finite number of local copies of prefetched services and mobile extensions and evicts the rest using an MFU policy (results for LRU replacement and exponentially decaying average MFU are similar but not shown.) For the other workloads (not shown), the results are qualitatively similar, but the cache size needed for full benefits is larger for the Squid-P workload and smaller for the Squid-C and BU-C workloads due to the differing number of services accessed by each of these workloads.

These experiments suggest that to take full advantage of client independence for improving availability, client and proxy virtual machines must be scalable to handle hundreds or thousands of simultaneously downloaded extensions in order to replicate a significant fraction of accessed sites. We examine the resource management challenges posed by such a workload in a separate study [4].

4.2 Network routing

In this section, we evaluate strategies that route around network failures. To simplify the analysis, we classify strategies into two broad categories: network re-routing and server replication and selection.

1. **Re-routing.** Techniques of this category still send requests to the service's origin server, but they may use alternate routes when failures occur. Examples of re-routing techniques include dynamic routing [15] and overlay networks [26]. In the terminology of this paper, these techniques address in-middle failures, but will be ineffective against near-source and near-destination failures.

2. **Server replication and selection.** This category of techniques directs requests to replicas of the origin servers when the origin servers are unreachable. Several file systems [25] and databases [19] provide replicated servers to handle failures in distributed environments. In the context of the Web, mirror site with "manual failover", as well as replicated servers with anycast [2, 8, 35] can support server replication. This class of techniques can resolve near-destination and in-middle failures but is ineffective against near-source failures.

As in our analysis of client independence techniques, we abstract implementation details of routing-based techniques and focus on bounding improvements that they may provide. Several factors may limit these improvements in practice. For re-routing strategies, overheads include the failure detection time and route switching time. For server replication and selection, there are costs to maintain extra replicas and overheads to select alternative servers. These overheads vary for different implementations and may vary for different services (e.g. depending on failures, consistency, and semantics). Therefore, as with client-independence techniques, clients may experience sessions handled by re-routing or server replication as "degraded" with the significance of the deterioration varying on a service-by-service and implementation-by-implementation basis.

Workload and methodology. We use the same workloads and similar methodology as for the client-independence experiments. We group requests into

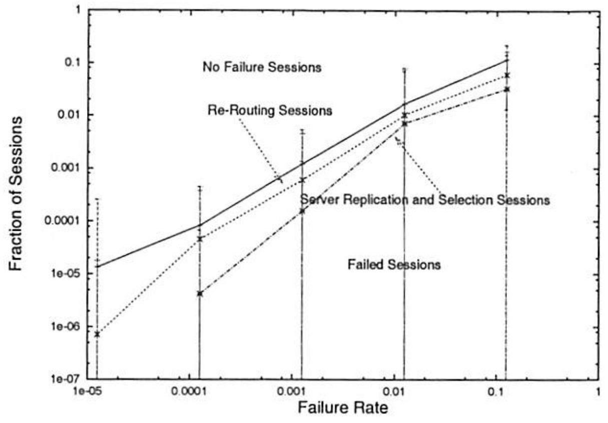


Figure 8: Session failure rate v. network failure rate (BU-P trace).

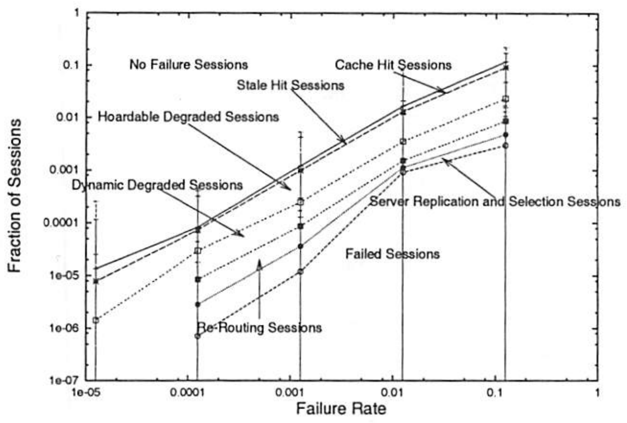


Figure 9: Session failure rate v. network failure rate (BU-P trace).

sessions, and classify each failure by its network location: near-source, in-middle, or near-destination. We run each experiment 25 times and plot the mean with 90% confidence intervals.

Results. In our first set of experiments, we vary the fraction of failures in each location category. These graphs are omitted due to space limits. Across a wide range of ratios, the findings are as expected: the fraction of failures that each class of techniques can handle varies in proportion to the fraction of failures assigned to a particular location category. For example, when in-middle failures account for 50% of all failures, techniques that avoid in-middle failures but not others can improve failure rates by about a factor of two. Given that experiments found significant fraction of failures at each location, Amdahl's Law limits improvements from routing based strategies that do not address failures in all three locations.

Figure 8 shows the sensitivity of these results as we vary the network failure rate. As for the client-independence strategies, the relative improvements to failure rates provided by these techniques remains stable over a wide range of underlying failure rates.

4.3 Combined Techniques

Client-independence techniques are limited by compulsory misses and installation time, and re-routing techniques are limited by near-source failures. Since these techniques fail in different circumstances, they may be combined to reduce system unavailability.

For example, Figure 9 shows session failure rates under a combined scheme in which failures are masked by caching, prefetching, and active objects and in which prefetching and installation of active objects use anycast to access replicated servers. This combined approach thus masks all failures except near-source failures during prefetching or active object installation time. Due to a bug in the simulator, Figure 9 shows results for *install_time* = 1; Figure 5 suggests that these results will be relatively insensitive to increases in *install_time*. Overall improvements for this combined scheme are factors of 117, 100, 18.2, and 24.5 for network failure rates of 0.0125%, 0.125%, 1.25%, and 12.5%, respectively. This relatively wide improvement range appears to be due to experimental variation magnified by the small number of failure events observed in the simulations.

5 Conclusions

Although Internet services can deploy highly available servers, deploying highly available *services* remains problematic due to connectivity failures. A typical client may not be able to reach a typical server for 15 minutes per day.

In this paper, we develop a network failure model and an evaluation strategy for studying broad classes of techniques for coping with connectivity failures. Both client-independence and routing-based techniques can significantly improve availability, and the techniques can be combined to improve availability by as much as one to two orders of magnitude.

References

[1] Network Appliance. Internet content adaptation protocol (icap). DS-2326, June 2000.

[2] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, N. Shah, and Z. Fei. Application Layer Anycasting. In *Proc. IEEE INFOCOM'97*, 1997.

[3] P. Cao, J. Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proc. of Middleware 98*, 1998.

[4] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Nayate, A. Razzaq, and A. Sewani. Resource management for

scalable disconnected access to web services. In *10th International World Wide Web Conference*, May 2001.

[5] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Traces. Technical Report TR-95-010, Boston University Department of Computer Science, April 1995.

[6] D. Duchamp. Prefetching Hyperlinks. In *Proc. of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[7] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[8] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar. A Novel Server Selection Technique for Improving the Response Time of a Replicated Service. In *Proc. of IEEE Infocom*, March 1998.

[9] J. Gwertzman and M. Seltzer. The case for geographical pushcaching. In *HOTOS95*, pages 51–55, May 1995.

[10] M. Harchol-Balter. The Effect of Heavy-Tailed Job Size Distributions on Computer System Design. In *Proc. of ASA-IMS Conf. on Applications of Heavy Tailed Distributions in Economics, Engineering and Statistics*, June 1999.

[11] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 2nd edition, 1996.

[12] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[13] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, and M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proc. of the Fifteenth ACMSymposium on Operating Systems Principles*, December 1995.

[14] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[15] K. R. Krishnan, R. Doverspike, and C. Pack. Improved Survivability with MultiLayer Dynamic Routing. *IEEE Communications Magazine*, 33(7), July 1995.

[16] T. Kroeger, D. Long, and J. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[17] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental Study of Internet Stability and Backbone Failures. In *FTCS99*, June 1999.

[18] D. Li and D. Chariton. Scalable Web Caching of Frequently Updated Objects Using Reliable Multicast. In *Proc. of the Second USENIX Symposium on Internet Technologies and Systems*, pages 1–12, Oct 1999.

[19] A. Moissis. SYBASE replication server: A practical architechture for distributing and sharing corporate information. Technical report, SYBASE Inc, March 1994.

[20] A. Myers, P. Dinda, and H. Zhang. Performance Characteristics of Mirror Servers on the Internet. In *Proc. of IEEE Infocom*, 1999.

[21] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[22] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proc. of the ACM SIGCOMM '96 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 22–36, July 1996.

[23] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics.* PhD thesis, University of California, Berkeley, April 1997.

[24] J. Pitkow and P. Pirolli. Mining Longest Repeating Subsequences to Predict World Wide Web Surfing. In *Proc. of the Second USENIX Symposium on Internet Technologies and Systems*, pages 139–150, Oct 1999.

[25] Mahadev Satyanarayanan, Member, IEEE, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[26] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The End-to-end Effects of Internet Path Selection. In *Proc. of ACM SIGCOMM '99*, pages 289–299, September 1999.

[27] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 172–183, December 1995.

[28] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proc. of the Nineteenth International Conf. on Distributed Computing Systems*, May 1999.

[29] G. Tomlinson, H. Orman, M. Condry, J. Kempf, and D. Farber. Extensible proxy services framework. IETF-Draft draft-tomlinson-epsfw-00.txt, IETF, July 2000. Expires January 11, 2001.

[30] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Naming: Flexible Location and Transport of Wide-Area Resources. In *Proc. of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[31] D. Wessels. Squid Internet Object Cache. http://squid.nlanr.net/Squid/, August 1998.

[32] R. Wolff. Poisson Arrivals See Time Averages. *Operations Research*, 30(2):223–231, 1982.

[33] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-Based Analysis of Web-Object Sharing and Caching. In *Proc. of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[34] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *Proc. of the Seventeenth ACM Symposium on Operating Systems Principles*, December 1999.

[35] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proc. of the 1997 USENIX Technical Conf.*, January 1997.

[36] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, AT&T Center for Internet Research at ICSI, http://www.aciri.org/, May 2000.

PRO-COW: Protocol Compliance on the Web—A Longitudinal Study

Balachander Krishnamurthy
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932
bala@research.att.com

Martin Arlitt
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
arlitt@hpl.hp.com

Abstract

With the recent (draft) standardization of the Hypertext Transfer Protocol - HTTP/1.1 protocol on the Web, it is natural to ask what percentage of popular Web sites speak HTTP/1.1 and how *compliant* are these so-called HTTP/1.1 servers. We attempt to answer these questions through a series of experiments based on the protocol standard. The tests were run on a comprehensive list of popular Web sites to which a good fraction of the Web traffic is directed. Our experiments were conducted on a global extensible testing infrastructure that we built to answer the above questions. The tests were carried out over a period of 16 months and were repeated thrice during the period. Our results show reasons for concern on the state of HTTP/1.1 protocol compliance: some servers do not properly support basic features such as the `HEAD` method, while many popular servers do not support some of the key features (such as persistent connections) that were added in HTTP/1.1. Perhaps most alarming of all, some servers crashed during testing. As a result we believe that small (but significant) changes to the wording of the protocol specification are required.

1 Introduction

With the recent IETF draft standardization of the Hypertext Transfer Protocol—HTTP/1.1 protocol in RFC 2616, it is natural to expect a significant number of clients (browsers), proxies, and servers would be upgraded to adhere to the HTTP/1.1 protocol requirements. In earlier work, we examined the key differences between HTTP/1.0 and HTTP/1.1 [KMK99]. In this work, we report on a longitudinal study that monitored the level of penetration in the Internet of the HTTP/1.1 protocol and the level of protocol *compliance* on the Web (PRO-COW). Many servers (and clients) are configurable and some have already reverted to *disabling* certain features (such as persistent connections) for a variety of reasons.

A wrong way to measure compliance is contacting a Web server and simply checking if the version number field in the response header is HTTP/1.1. This ignores the inherent complexity of the Web (e.g., presence of proxies and gateways), mis-interprets the protocol version number in the response header, and assumes that the server is actually fully compliant based on the protocol version it reports. The question, however, is important and finding a proper answer based on a clear understanding of the Web infrastructure and the HTTP/1.1 protocol would be useful for several reasons. Knowledge of the number of servers actually running fully compliant HTTP/1.1 servers would give us a measure of the protocol adoption rate. If the number is low, it permits us to examine the reasons for the low adoption rate. Repeating the study periodically can be handy to know if future changes are warranted and the speed with which they should be introduced. If compliance with certain features is low, it could have an impact on introduction of similar new features. Web sites are the visible front end for e-commerce companies requiring them to avoid server failures. As we will see later in the paper, failure to adhere to some of the compliance requirements could lead to serious difficulties. An estimate of the traffic that is end-to-end HTTP/1.1 helps us quantify benefits (or disadvantages) of the changes to the protocol that cannot be gleaned by comparing the protocol version numbers.

There are different audiences for this paper. Server implementors may be interested in seeing the level of compliance of their product in practice. Some servers have front ends that distort and alter the semantics of the response leading to potentially incorrect conclusion about the compliance level of that implementation. Protocol designers can see what ideas actually get reflected in usage. Much debated additions to the protocol may not be implemented properly or may be turned off via configuration options in practice. Knowing the realities would help them deal with evolving the protocol. Those interested in learning about compliance issues of protocols in general could see what features are hard to get right and why. Web site administrators deciding on moving to the new version of the protocol can see if a reasonable number of popular sites are running HTTP/1.1. However, if many of the servers running HTTP/1.1 are not compliant or if several of the new features are not enabled, they may want to wait.

Given the enormity of the Web and the complexities of the new version of the protocol (the specification nearly tripled in size between HTTP/1.0 and HTTP/1.1), it is not easy to test the compliance of deployed Web servers. Fortunately however, our task is somewhat simplified for three reasons. Prior work [KMK99] (in which one of the authors of this paper was involved) examined the key differences between HTTP/1.0 and HTTP/1.1, categorizing and identifying the important changes in the protocol. Partitioning of the changes helped isolate the features and enabled the rapid construction of the set of compliance tests. Secondly, although there are tens of millions of sites, only a small fraction of sites attract a significant percentage of traffic. A plausible list of these sites is available from a set of survey sites. Thirdly, access to packet traces from a large ISP helps us verify the accuracy of the survey sites, calibrate the set of popular servers, and determine which (new) HTTP headers are commonly used.

Instead of designing a simple experiment that only addresses the issue of compliance when tested from a client to an origin server we have designed a global experimental infrastructure infrastructure spanning multiple countries and a mix of sites (educational, commercial, and soon residential sites connected via ISPs). The infrastructure is used periodically to study the evolution of protocol compliance. We conducted the compliance tests over a period of 16 months roughly every six months. Along the way several popular sites moved from HTTP/1.0 servers to HTTP/1.1 servers.

We have several significant findings. Over a 16 month period, the percentage of servers claiming to be HTTP/1.1 compliant increased with little improvement in the percentage of servers that were actually compliant. Some popular Web sites failed even the most basic compliance requirements. A significant number of servers have turned off a number of the key HTTP/1.1 improvements. Several servers crashed during our testing. Compliance does not necessarily improve over time. Many of the problems appear to be caused by incorrect server configurations or poorly implemented plugins or filters. A number of implementors have already utilized the results of our study. For example, a prominent browser changed their default behaviour on persistent connections and a server vendor fixed a serious security hole as a result of an earlier version of our study (we notified them discreetly and do not divulge the specific problem here).

The rest of this paper is divided as follows: Section 2 presents motivation for the move to HTTP/1.1 and Section 3 discusses related work in this area. Section 4 presents our compliance testing methodology while Section 5 describes the actual experiment performed to measure the compliance. Section 6 discusses the software used to test and the environment in which the experiment was conducted. Section 7 discusses the results of the experiment. We conclude with a summary of the paper and a discussion of future work.

2 HTTP/1.1 Protocol

Along with the astounding increase in network traffic of HTTP packets, several problems were discovered in the HTTP/1.0 protocol. There was no official standard for the HTTP/1.0 protocol though there were various implementations. The latest version of the Hypertext Transfer Protocol HTTP/1.1

was standardized in June 1999 (in RFC 2616 [ea99]) after over four years of discussion in the IETF HTTP Working Group. A primary motivation in coming up with a new version of the protocol was to fix several known weaknesses in HTTP/1.0. However, during the process of developing the standard, several intermediate "HTTP/1.1" implementations of servers and clients began showing up on the Web.

As part of the standardization effort of HTTP/1.1, reports on several interoperable implementations of the components (clients/browsers and servers) had to be submitted to the W3C–World Wide Web consortium. W3C forum reports describing the features supported by various implementations are available in [For]. The information in the forum reports are submitted by the various individuals/organizations who developed the components. The forum report lets developers indicate the subset of features of the protocol they have implemented and if they have tested the features.

The testing that is reported in the forum is done by the developers themselves but occasionally components are made available for others to test. Some of the servers on the forum are very popular on the Web (e.g., Apache, Netscape-Enterprise, and Microsoft-IIS) while others are experimental servers often used in the research community. There is a reasonable amount of variance in the set of features implemented in this collection and compliance testing cannot be done by just testing the subset presented in this forum. Given that there are millions of sites running servers with different configurations, it becomes important to broaden our compliance testing base and methodology from that used in the forum.

3 Related work

Measurement of protocol compliance is not entirely novel but we know of no other independent testing in the HTTP arena. Partially, this is due to the relative recency of the protocol and HTTP/1.1 is the first upgrade since HTTP/1.0. Earlier versions, such as as HTTP/0.9 and HTTP/1.0 were never formally standardized. Formal testing of non-standardized versions would not have been very helpful. However, the deficiencies found in the implementations of HTTP/1.0 and the prematurely (mis-)labeled HTTP/1.1 helped in the clarification of the actual HTTP/1.1 specification. Forum reports [For] from implementors and interoperability testing conducted via the Web consortium aided in finding some problems. In the commercial arena there are more *claims* than actual evidence of compliance: many products claim compatibility or compliance with HTTP/1.1 to improve their marketability.

4 Methodology

In this section, we present the design decisions of our experiment. We had to decide if the compliance tests could be run locally and if not, how to come up with a canonical set of test sites. We then had to assemble a testing infrastructure, identify testing software, and isolate the features to be tested based on the protocol draft standard. We discuss the various tradeoffs and the reasons behind the choice of the approach we took.

One approach to test compliance would have been to choose just a handful of popular servers (e.g., Apache, Netscape, Microsoft-IIS) and run our tests directly on the software in a lab environment. There are many reasons for *not* doing this. First, it is extremely difficult to test all of the software configurations (we counted nearly 60 different configurations of the Netscape server and close to 700 different configurations of the Apache server in a recent packet trace from a large ISP). Secondly, different answers might result based on where the tests originate (although we did not expect the results to vary based on client location, but we wanted to confirm this hypothesis). Thirdly, it is more interesting to see what installed servers in the field do than a particular binary release with a fixed configuration. Fourth, it is not possible to obtain the source and binaries of all the servers. Several popular sites run proprietary servers designed just for their organization. Finally, we wanted to test under real world conditions – our requests would be like any other Web request. As will be seen later, this decision was well warranted, since some server implementors were surprised to see the circumstances under which their server was being used.

Additionally, even the information about the server returned in the HTTP response (in the `Server:` header) didn't always include any configuration information. This leads to anomalies such as the same feature implemented correctly in some sites and not so in others, although both sites apparently ran the "same" server instance. Relying on the server identifier would thus be a mistake leading to wrong conclusions.

It has been well known for a while that a small number of Web sites attract a significant portion of the Web traffic. This has led to the compilation of popular Web sites which has significant economic value to those sites (revenue from advertisement). Rating sites such as MediaMetrix [Med], Netcraft [Net], and Hot100 [Hot] offer statistics on popularity. In addition, some compilations of well known corporations such as Fortune 500 [For99] and Global 500 [Glo98] provide lists of globally known corporations some of which presumably attract a significant amount of Web traffic. Our combined list contained more than 500 unique Web sites.

Each of the survey sites has a different way of gathering their data and none of them have made their methodology transparent enough for independent testing. Some even say that they deliberately withhold this information to ensure surveyed sites do not misuse it to alter their rankings. Hot100 claims to survey 100,000 users (40% of whom are outside the USA). They claim to gather data at "strategic" points on the Internet (not at the browser or server) which they then sift through. Mediametrix claims to use a 50,000 user population. Our longitudinal study showed that throughout the 16 month period in which our study was conducted, over 40% of the top 150 unique sites from the combined MediaMetrix [Med], Netcraft [Net], and Hot100 [Hot] ranking lists were the same.

If we are interested in testing compliance of the HTTP/1.1 protocol, it would make sense to examine the servers running on popular Web sites. We performed an initial test to determine which sites were using HTTP/1.0 and which sites were claiming to be HTTP/1.1 compliant. All of our remaining tests were performed using only the list of sites that claimed to be HTTP/1.1. We contacted each one of these sites from a variety of locations in the world to ensure that the client location didn't introduce any bias. Our clients were either the *httperf* [MJ98] tool or handwritten C code imitating basic aspects of a browser and saving the response headers returned. Part of the reason for testing from a variety of places is to extend the tests in the next round to go through proxies (rather than directly from client to server as in this work). We also considered some of the W3C forum reports [For] submitted to the W3C consortium by various server and client implementors to see if the features that were reported as implemented and examined in the interoperability tests are indeed compliant. We included a subset of tests that most of the implementors had conducted.

Next, it is important to measure the origin server's compliance without having to worry about the influence of proxies, gateways, and tunnels in the path. Proxies, depending on if they are transparent or non-transparent may modify the requests and alter some of the headers in either direction. The response from the server would not be seen directly by the testing client; only the response sent by the proxy. Our knowledge of our local networks allowed us to avoid both non-transparent and transparent proxies at the client sites. However, even when we send requests directly from clients to origin servers, it is possible for an intermediary in front of the server to intercept the request (we noticed several such cases in our test).

In terms of verifying compliance, we primarily relied on the protocol standard [ea99]. The protocol specification has three classes of compliance by clients, proxies, and servers for features: MUST, SHOULD, and MAY [Bra97]. The HTTP/1.1 specification states that a server implementation that fails to satisfy one or more MUST requirements is *not* compliant. If it satisfies all MUST and SHOULD requirements it is *unconditionally* compliant and if it meets all MUST but not all SHOULD requirements it is *conditionally* compliant.

It should be noted that our tests are *not* simply a test of the MUST, SHOULD, and MAY requirements of the HTTP/1.1 draft standard. Instead, we have divided the tests into categories based on importance to the overall Web infrastructure. While some servers may have consciously not complied with some requirements or turned off some features (since they are *not* a MUST requirement), we may still highlight that fact. The goal of our experiment was to both classify the servers in terms of compliance and also speculate on the reasons for any non-compliance. While we have not tested every feature of the protocol for compliance, we have prioritized and tested some of the key features. The primary reason we did not develop a complete test suite for HTTP/1.1 compliance is that it would be extremely difficult to automate some of the tests without having specific knowledge of the design of each site under test (an example of this is provided in Section 7.2.3. Our testing model is extensible—other features simply require extensions to the script which can be plugged into our testing and analysis infrastructure. This enables continued testing over the long haul as more server sites move to HTTP/1.1. Our continued testing also helped us to monitor changes to the servers over time. We also kept our tests free of biases such as time of day and location of clients.

5 Compliance experiment

The actual compliance experiment involved extracting important features from the HTTP/1.1 protocol specification as presented in RFC 2616, the HTTP/1.1 draft standard. Several of the features of 1.1 are carried over from HTTP/1.0 since all HTTP/1.1 servers have to accept HTTP/1.0 style requests. We divided our experiment into three categories: important features in the protocol specification (all of which are MUST conditions; i.e., the implementation is not compliant otherwise), features that we believe are significant additions in HTTP/1.1, and features that are not mandatory in servers yet are considered useful in evolving the protocol. We expect every compliant server to meet the tests of the first category, most to meet the second category tests, and expect significant variance in compliance for the third category tests. Our testing infrastructure can be easily extended to do other compliance tests by augmenting the scripts and reusing the largely automated analysis process.

5.1 Category One tests

In the first category of the experiment we tested `GET` and `HEAD` methods with modifiers as warranted, and tested for the absence of the required `Host` header. We expect these tests to succeed in *any* compliant HTTP/1.1 server. Servers that do not implement the above features correctly are presumably invalidly labeling themselves as HTTP/1.1. It should be noted that the version number in an HTTP message is a hop-by-hop header (as opposed to an end-to-end header) and since our tests are directly from client to origin server, we get exactly the version number the origin server claims it implements.

A vast majority of all HTTP requests made to Web servers are `GET`, the basic way to request a resource on the Web. The `HEAD` method requests that only metadata about the resource be returned and is often used to debug servers. Neither of these methods are new in HTTP/1.1, nor has their behaviour changed significantly. Use of modifiers with `GET` (such as `If-Unmodified-Since`), however, are new and thus included in our tests. These tests are to verify that servers respond with (the new response code) `412 Precondition Failed`, when the precondition fails.

The `Host` header was added to slow down the depletion of IP addresses, due to a rush to obtain vanity URLs (such as `www.foo.com`) and HTTP/1.0 requests not passing the hostname of the request URL. Rather than change the request line format (which would cause massive configuration difficulties), a new (`Host:`) header was *mandated* to be present in every HTTP/1.1 request message. If a HTTP/1.1 request message does not have the `Host:` header it **must** be rejected.

5.2 Category Two tests

The second category of tests consists of important features that have been added to HTTP/1.1. A significant amount of discussion ensued on some of these features during the over four-year development of HTTP/1.1. In the case of a server mis-implementing or partially implementing features tested in this category, we would be curious to know why. Unlike Category One tests, where errors simply imply non-compliance, this category includes tests of features that servers are permitted to selectively implement. There is a general expectation that a HTTP/1.1 server would implement these features. The tests that we include in this category are handling of persistent connections, pipelining, and range requests.

Introduction of persistent connections was a major innovation in HTTP/1.1. In HTTP/1.0, connections lasted just for for a single request/response exchange. This had both a deleterious effect on user perceived latency and the server (each request required TCP setup and teardown) as well as the network (in terms of the additional packets). Most HTTP transactions are short and the TCP handshakes consumed a good chunk of the overall time. For pages with a dozen embedded images (a figure that is relatively common), multiple TCP setups and teardowns were needed. Mogul and Padmanabhan suggested the introduction of persistent connections based on an experimental study [PM95]. Persistent connections are the *default* in HTTP/1.1, though servers or clients could close the connection after the first exchange. In fact, downloading all the embedded images in a single persistent connection (labeled as perfect persistence [KW00]) has the best performance.

Pipelining permits clients to send a stream of requests in a pipeline without waiting for any response from the server. The round trip time of waiting for the acknowledgments of the previous request is eliminated. The server however sends the responses in the order of the requests received. Further studies [ea97] revealed that persistent connections without pipelining could in some cases worsen the performance. In some cases multiple parallel non-persistent connections were found to be better but this came at a cost (minimal to the browser, higher to the server in order to deal with multiple simultaneous connections from each client). Persistent connections with pipelining provided the best combination to reduce latency and the overall number of packets.

Recently, there has been anecdotal evidence (in discussions and mailing lists) that some sites are turning off persistent connections. If it were true, one of the key perceived advantages of HTTP/1.1 (to the network in terms of reduced packets and to users in terms of latency) would turn out not to have been realized. We tested if servers permitted the basic ability to hold the connection open beyond a single connection and then a separate test of its ability to handle pipelined requests.

Another innovation in HTTP/1.1 was the ability to request byte ranges of resources rather than the full contents. There are several reasons for this, including efficiency, such as requiring just the tail of a growing resource, prefetching the headers of resources of certain content-type (such as *gif, jpeg*) to begin outlining images before actually fetching the image, etc. Recovering from aborted connections and transfers is eased by range requests. When parts of the resources are cached, only the missing parts need to be obtained.

5.3 Category Three tests

The third category of tests include minor issues that servers should normally be compliant with. Consequences of non-compliance here are less severe than the first two categories.

5.3.1 Additional method tests

The HTTP/1.1 specification clearly indicates that all general purpose servers MUST implement `GET` and `HEAD` methods. Support for other methods are optional but a server implementing other methods must conform to the specification (Section 5.1.1 of [ea99]). Thus, our tests of conformance is one of proper compliance *if* other methods are implemented. The OPTIONS method indicates the capabilities of the origin server. With a resource specified, any optional features applicable to that resource alone is returned. The TRACE method purely runs a loopback test of the message included in the request and is simply a way to see if the server received exactly what was sent from the client. The server is supposed to return the request it received in the response body.

5.3.2 Expect/Continue mechanism

To prevent clients from needlessly sending large bodies in PUT/POST requests that might not be accepted by a server, HTTP/1.1 introduced a mechanism by which clients could check with the server beforehand. A client would send just the header (without a body but with a content length indicator) including a request header `Expect: 100-Continue`. If the server is willing to accept the request it would reply with a `100 Continue` status response and then the client can send the body; otherwise the server can send a `401 Unauthorized` or a `417 Expectation Failed` response.

5.3.3 Conditional requests

HTTP/1.1 introduced several new conditionals to improve the caching model. Instead of the simple Last-Modified timestamp check that HTTP/1.0 provided in the `GET If-Modified-Since` request, the presence of opaque strings in the form of Entity tags, permits a more general model. If several instances of a resources are maintained at the server and cached at a proxy, the proxy could check if any of its cached instances are current by including conditional headers such as `If-Match`. Additionally, an `If-Unmodified-Since` conditional permits a resource to be sent only if it has *not* changed since the indicated date.

When performing timestamp checks the format of the date string is an issue. HTTP applications have permitted three date formats RFC 822 (updated by RFC 1123), RFC 1036, and ANSI C's `asctime()`. While, HTTP/1.1 clients and servers have to accept all three formats for compatibility with HTTP/1.0, they can *only* generate the RFC 1123 format for representing date values in header fields.

5.3.4 Miscellaneous tests

Several new requests and responses have been added in HTTP/1.1. We examine a variety of method and header combinations for violations of size or incorrect headers. We also check for the server's ability to handle long and incorrect URLs in the request.

6 Testing software and environment

For testing we primarily relied on *httperf* [MJ98] – a performance measurement tool for HTTP. *Httperf* is useful for understanding Web server performance and for analyzing server features and enhancements. The tool has three logical components: the core HTTP engine, the workload generator, and the statistics collector. The latter two components can be configured at runtime via command line options.

We chose to use *httperf* as an analysis tool for several reasons. Since *httperf* already supports the HTTP/1.1 protocol, it saved us from having to provide our own implementation. Although *httperf* does not currently support all of the features that we needed for this study, the tool is available in source code form. This enabled us to modify the tool to issue the desired request headers and collect the appropriate statistics. Our extensions can be rolled back into *httperf* for others to use.

We tested compliance from a variety of places around the world from diverse organizations. Although this should not be necessary since a server should give the same answer no matter where the requests came from, we did this for two reasons. First, studying an origin server's compliance is just the first stage of our experiment. Additional experiments that test other artifacts of the Web require diversity of location to expose biases in the path between client and server (as in [KW00]). Second, if we noticed any dependency on the origin of requests (though the web sites were contacted using hardwired IP addresses) that would be of interest to explore.

When performing tests of such a large scale nature on several large sites, one has to be careful not to let the testing interfere with the normal workings of the site. We did this by identifying ourselves via the `From:` and `User-agent:` headers in every request we sent. We also sent our few test requests serially and just once.

A C program parsed the response headers in the output from each client site (converting them into integer and/or bitmap descriptors) and verified presence/absence of headers, gleaned values, and checked for appropriate headers.

Primary tests were run from the authors' respective organizations (AT&T Research located in New Jersey, USA, and Hewlett-Packard Labs in California, USA). Additional tests were run from the University of Kentucky in Lexington, Kentucky (USA), the University of Paris-Sud, Orsay, (France), the University of Western Australia (Nedlands, Western Australia), and a commercial site in Santiago (Chile). Even though we chose different organizations and locations, the same software was installed and used for all the tests. Each test was run once from each testing site. The same set of servers were contacted from each of the sites. The machines all ran different versions of the UNIX operating system.

Table 1: Breakdown of server software and protocol version

Vendor/Version	Jun 99	Nov 99	Sep 00
Netscape	34.8%	38.8%	38.1%
Microsoft	32.8%	30.9%	33.3%
Apache	28.2%	26.8%	25.3%
Lotus	2.7%	2.6%	1.9%
Zeus	0.4%	0.6%	0.3%
Others	1.1%	0.3%	1.1%
HTTP/1.0	27.0	16.2	7.5
HTTP/1.1	73.0	83.8	92.5

7 Results

7.1 Experiment details

Although the Apache server has a large lead in the server market (over 60% of the market according to Netcraft [Net]), a majority of the popular sites are running Netscape and Microsoft-IIS servers. Table 1 shows vendor-based distribution of servers. The most popular version of the top 3 servers are Netscape-Enterprise/3.x, Microsoft-IIS/4.0, and Apache/1.3.x. Since servers from just three organizations are used by over 95% of the most popular sites running HTTP/1.1, compliance can be improved significantly by ensuring that *all* configurations of these servers are fully compliant.

Table 1 also shows the increase in the number of popular servers that are claiming to be running HTTP/1.1 over the course of our study. While the table shows a significant increase in servers claiming to be running HTTP/1.1, our results indicate that improvement in compliance has not kept pace.

There are three important caveats to be kept in mind while interpreting our results. First, there are several popular sites that still (as of September 2000) use HTTP/1.0 servers. Among these are AOL, Excite, Yahoo, Amazon, and Altavista. In fact over 7% of the HTTP/1.0 servers we contacted are not Y2K compliant in their `Date` header format. Second, the popularity as measured by MediaMetrix [Med], Netcraft [Net], and Hot100 [Hot] is purely based on number of requests sent to these sites and *not* on the bytes of response. One could just as easily make a case that a top n listing based on bytes shipped from servers is a more important metric. If we are seeking bandwidth reduction through the use of new features in HTTP/1.1, byte-wise high volume servers are likely be to better targets. However, the lack of server logs from these top n sites does not give us a way of measuring this. Finally, there is evidence that pornographic sites are downplayed in surveys of sites. Thus it is possible that such sites didn't end up in our top n sites list. Additionally, such sites, given their propensity to include more images, often have a higher volume (bytewise) of response traffic than other sites. We have been able to verify this based on two separate sets of data: a packet trace from a large ISP where half a dozen porn sites running HTTP/1.1 had many more response bytes compared to the rest of the frequently visited sites, and a proxy log from a large content hosting ISP. We chose not to change our methodology to add such sites to our list.

7.2 Experimental results

In Section 5, we divided our tests into three categories. In this section, we examine the results for each of the categories, as well as for most of the set of tests that we perform. Our compliance tests focus on the extreme cases: determining which servers satisfy all MUST and SHOULD requirements (the unconditionally compliant servers); and determining which servers cannot satisfy one or more MUST requirements (the non-compliant servers).

The initial round of testing in June 1999 confirmed our hypothesis that the location of the request origin does not (in general) have an impact on the compliance. However, there were a few minor variations. We identified two potential causes for the variation in the results by client site. Although we attempted to contact the same server from each client (by utilizing the IP address rather than the host name of the server), some sites appear to use load balancers or other devices to distribute incoming requests among a cluster of (heterogeneous) servers. Furthermore, a number of sites were unavailable during several of our tests; since our results are calculated based on the number of sites that responded to a test (unless otherwise noted), this creates a small amount of variability in the computed percentages. In June 1999 approximately 2% of sites were unavailable; in the last two studies the number of unavailable sites was around 1%. Since the results varied only slightly depending on which client site is utilized, we use only

Table 2: Unconditional Compliance Results for Category One Tests (HPL Data). All figures are in percentage.

Date	GET	HEAD	Host	Pass All	Fail All
Jun 99	83.5	72.9	64.5	60.6	7.1
Nov 99	82.4	69.6	60.0	56.8	7.3
Sep 00	81.9	72.4	64.7	59.1	6.5

Table 3: Breakdown of Category One Test Result Ranges (HPL Data) All figures are in percentage.

	GET	HEAD	Host
Unconditional	81.9–83.5	69.4–72.9	60.0–64.7
Missing Headers	16.1–17.9	8.9–10.8	28.6–30.4
Not Compliant	0.2–0.4	17.7–19.6	6.6–9.6

one (the HPL site) for discussion purposes throughout the remainder of this section. Details on the other sites are available in [KA99].

7.2.1 Category One test results

In this first round of tests we examine three required features for HTTP/1.1: the `GET` and `HEAD` methods, and the `Host` header. We consider a server to be unconditionally compliant for these tests if it returns an appropriate status code (200 for `GET` and `HEAD` tests, and 400 for the `Host` test), and if the response includes the appropriate headers (e.g., `Date` and `Content-Length` or `Transfer-Encoding: chunked`).

The results of this analysis are shown in Table 2. The results reveal that across the three measurement periods the results were quite consistent, with around 82% of the sites under study unconditionally compliant with respect to the `GET` method; about 70% of the sites were unconditionally compliant for the `HEAD` method, while over 60% passed the `Host` test. About 60% of the sites under study were unconditionally compliant across all three tests, while around 7% of the sites failed all three tests for unconditional compliance. Table 2 reveals that the results were quite consistent across the studies conducted over a 16 month period. Note that even with the migration of several servers from HTTP/1.0 to HTTP/1.1 during the testing period and the introduction of new server versions, there has been little improvement in compliance. In order to understand why so many servers failed to pass one or more of these three basic tests we examine each test separately.

Table 3 provides a more detailed breakdown of the Category One results. The ranges represent the minimum and maximum observed results over the entire duration of the study. As we have already shown, most servers are unconditionally compliant with respect to `GET` requests. Between 16 to 18% of the servers did not include either a `Content-Length` header or a `Transfer-Encoding:chunked` header to indicate to the client the length of the message body. Due to this omission these servers are characterized as conditionally compliant. Several servers failed the `GET` compliance test either by returning an incorrect status code or an incorrectly formatted `Date` header.

Around 70% of the tested servers were unconditionally compliant with respect to `HEAD` requests. Approximately 9% of the tested servers did not include the same entity headers that were seen in response to the `GET` request. Thus, these servers are deemed to be conditionally compliant. The more intriguing result is that almost 18% of the tested servers failed the compliance test because they returned a status 500 response rather than the expected 200 response.

Table 3 indicates that close to two-thirds of the servers fulfilled all requirements when responding to a request that did not include a `Host` header. Nearly 30% of the tested servers did not include either a `Date` header or one of `Content-Length` or `Transfer-Encoding: chunked`. These servers are considered to be conditionally compliant. Between 6 and 9% of the servers were not compliant, as they did not require the `Host` header to be present.

To determine whether a specific type of Web server was responsible for the unusual behaviour we observed, we analyzed the data by the type of server. The results for the five most common servers seen in our tests are shown in Table 4. These five server versions accounted for an average of 88.7% of all the servers seen in our tests over the testing period. Each server has three rows associated with it representing data from Jun '99, Nov '99, and Sep '00 respectively.

Table 4 indicates that two of the top five servers (Apache/1.3 and Apache/1.2) were unconditionally compliant on almost every site under study; in the few cases where these servers were not unconditionally compliant, the cause was usually a missing header, perhaps the result of the (mis)configuration of that particular server. Microsoft-IIS/4.0 ranked third in the percentage of servers that passed all three tests for unconditional compliance, trailing the Apache servers by about 10%. Most of the remaining Microsoft-IIS/4.0 servers did not issue the expected response headers. None of the Apache/1.3, Apache/1.2 or Microsoft-IIS/4.0 servers failed all three tests for unconditional compliance.

The results are less positive for the Netscape-Enterprise 3.5 and 3.6 servers. None of these servers passed all three of our tests for unconditional compliance. In fact, over 15% of the Netscape/3.5 servers and over 24% of the Netscape/3.6 servers failed all three unconditional compliance tests. This observation suggests that perhaps there is a problem with the configuration of these servers at some sites. These results also suggest that certain types of Web servers are responsible for much of the absence of compliance we noted earlier in this section. A somewhat discomforting observation is that the Netscape/3.6 server appears to be less compliant than its ancestor, the Netscape/3.5 server; i.e., things do not necessarily improve over time. Reviewing the results in Table 4 suggests that the biggest change between these versions occurs with the `GET` requests. While nearly 70% of the Netscape/3.5 servers were unconditionally compliant on this test (the remaining 30% were missing a `Content-Length` or `Transfer-Encoding: chunked` header), only 56% of the Netscape/3.6 servers passed unconditionally (the remainder were missing a length header). At this time we do not know why this change has occurred. Both the Netscape/3.5 and 3.6 servers did quite poorly on the `HEAD` test, with only around 21% and 27% passing the unconditional compliance tests respectively. The remaining Netscape 3.5 and 3.6 servers were either missing the expected headers or returned an incorrect status code. All of the Netscape 3.5 and 3.6 servers failed the `Host` test for unconditional compliance. About 80% of the 3.5 servers and 85% of the 3.6 servers were missing both a `Date` and a length header, while approximately 20% of the 3.5 servers and 14% of the 3.6 servers were not compliant, as they did not require a `Host` header to be present.

At this time, there are only a few popular server sites that run IIS/5.0 and Netscape 4.x and so there is insufficient data to draw any conclusions. We plan to continue to monitor changes in the distribution of servers used by popular sites, and analyze the compliance of the most common versions.

Even though we found that certain types of Web servers are responsible for a lot of the non-compliant behaviour we observed, the results in Table 4 also indicate that there is a lot of variability in the degree of compliance even among more homogeneous groupings. This suggests that the configuration (including the use of plugins) may have a significant role in determining how compliant a server is and validates the methodology of our testing actual server sites rather than the server software.

The results in Table 2 revealed that 40% of the servers failed one or more of the Category One tests for unconditional compliance to the HTTP/1.1 specification. The most common reason for failure was the lack of appropriate response headers.

Perhaps a more significant observation is more than 20% of the servers tested were not compliant (i.e., they failed a MUST condition) on at least one of the three basic functionality tests. 3% of the servers were not compliant on two of the tests. These servers should definitely not claim to be HTTP/1.1 applications.

7.2.2 Category Two test results

The next set of tests examined some widely discussed enhancements to HTTP/1.1—server support of persistent connections, pipelining, and range requests. Although maintaining a persistent connection is supposed to be the default behaviour of an HTTP/1.1 application, it is a SHOULD requirement and thus a server can be conditionally compliant without maintaining persistent connections. However, servers have to send a `Connection: close` header to indicate non-persistence. The Range feature is a MAY level requirement; servers can always send the complete response.

Table 4: Breakdown of Category One Results by Server Type (HPL Data)

Server	Date	%of Svrs	GET(%)	HEAD(%)	Host(%)	Pass All(%)	Fail All(%)
Apache/1.2	6/99	9.1	100.0	100.0	97.8	97.8	0.0
	11/99	6.3	96.9	100.0	96.9	93.8	0.0
	9/00	2.4	100.0	100.0	100.0	100.0	0.0
Apache/1.3	6/99	18.1	100.0	96.7	100.0	96.7	0.0
	11/99	19.7	99.0	99.0	99.0	98.0	0.0
	9/00	21.8	98.4	98.4	98.4	98.4	0.0
IIS/4.0	6/99	32.5	89.7	98.0	98.2	87.3	0.0
	11/99	30.7	89.7	98.1	98.1	88.4	0.0
	9/00	30.2	90.2	98.9	96.0	89.0	0.0
Netscape/3.5	6/99	14.0	70.8	22.2	0.0	0.0	16.9
	11/99	13.8	71.4	20.3	0.0	0.0	14.5
	9/00	6.2	66.7	22.2	0.0	0.0	13.9
Netscape/3.6	6/99	12.8	50.8	29.2	0.0	0.0	27.7
	11/99	21.7	58.2	23.8	0.0	0.0	23.9
	9/00	27.0	60.2	27.9	0.0	0.0	20.8

Table 5: Breakdown of Category Two Results (HPL Data)

Date	Persis-tence	Pipe-lining	Range	Pass All	Fail All All
Jun 99	71.9	70.6	52.3	43.9	20.6
Nov 99	71.5	64.8	54.9	41.0	21.2
Sep 00	74.3	66.8	54.7	42.8	20.8

Table 5 reveals that many of the servers supported at least one of the Category Two features. Over 70% of the servers supported persistent connections. We suspect that many remaining sites had persistent connections disabled by the administrators. Slightly fewer sites allowed the client to pipeline requests. Only about half of the servers supported Range requests properly. Only 40% of the sites supported all three Category Two features, while 20% of the sites supported none of these features.

The results in Table 6 reveal that the server type has less effect on the results than was the case for the Category One tests. Almost all of the Apache/1.2, Apache/1.3 and Microsoft-IIS/4.0 servers supported persistent connections and pipelining. Only about half of these servers supported the Range requests which is the primary reason why so few of these server passed all three Category Two tests. Fewer Netscape-Enterprise/3.5 and 3.6 servers supported persistent connections and pipelining than ranges. We have evidence that suggests there is a problem with the persistent connection implementation in instances of some servers (e.g., the server sends a `Connection: keep-alive` header with the response, then closes the connection without allowing the client to issue any subsequent requests).

We next examine the number of servers that (unconditionally) supported all six features. Table 7 shows that around 30% of the servers under study passed all six tests, while 7% of the servers were either conditionally compliant or not compliant on all six tests. Not shown in the table are the following figures: 15% of the Netscape-Enterprise/3.5 servers (N-E/3.5) failed all six tests (i.e., they supported zero features) and none supported all six features. Over 20% of the Netscape-Enterprise/3.6 servers (N-E/3.6) failed all six tests and none supported all six features. Approximately 50% of the Apache/1.2, Apache/1.3 and Microsoft-IIS/4.0 servers supported all six features.

7.2.3 Category Three test results

In this section we tested the servers to determine whether they supported nine other suggested/recommended but lesser known features of

Table 6: Breakdown of Category Two Results by Server Type (HPL Data)

Server	Date	Persistence(%)	Pipelining(%)	Range(%)	Pass All(%)	Fail All(%)
Apache/1.2	6/99	89.1	89.1	52.7	43.5	10.9
	11/99	90.6	90.6	46.9	40.6	3.1
	9/00	100.0	92.9	50.0	50.0	0.0
Apache/1.3	6/99	87.0	87.0	51.1	47.8	9.8
	11/99	89.0	89.0	54.0	51.0	8.0
	9/00	89.0	88.2	52.8	48.8	7.9
IIS/4.0	6/99	87.9	87.3	52.4	52.4	12.7
	11/99	85.9	85.8	55.8	54.5	12.8
	9/00	86.8	86.7	52.6	49.7	11.4
Netscape/3.5	6/99	41.1	38.4	67.2	37.5	30.6
	11/99	40.0	28.1	68.6	24.6	30.4
	9/00	44.4	31.3	63.9	27.8	36.1
Netscape/3.6	6/99	41.5	35.4	47.7	35.4	52.3
	11/99	50.0	33.6	54.6	32.7	44.6
	9/00	54.2	32.3	57.4	31.6	40.7

HTTP/1.1. Unfortunately many of these features are not straightforward to test. For example, POST requests may not be allowed for certain objects. In such a situation we can test that the server properly rejects the request, but we cannot test whether the server would properly handle a POST request.

Table 8 shows the results of our Category Three tests. The ranges shown in Table 8 indicate the minimum and maximum values seen over the three measurement periods. We observed that a significant number of servers returned non-compliant responses in some of our Category Three tests. For example, some servers return a status 200 response (along with a Web page that indicates an error has occurred) to a request for an incorrect/non-existent URL. This is due to the server configuration rather than the server implementation. The most significant occurrences of non-compliant responses happened with our "POST with `Expect: 100-Continue`" test and our `If-Unmodified-Since` tests. In the `Expect: 100-Continue` test we observed that many sites did not issue a response to the request. Since the number of sites that did not respond was much higher than normal (27–30% of sites versus an average of 1-2% for the other tests), we speculate that most of these sites are simply waiting for further information from the client. Most of the servers we tested do not appear to understand the `If-Unmodified-Since` header, and as a result of ignoring it return a non-compliant response. All of

Table 7: Breakdown of Category One and Two Results (HPL Data)

Date	Pass All Tests(%)	Fail All Tests(%)
Jun 99	31.1	7.1
Nov 99	29.8	7.3
Sep 00	31.4	6.5

Table 8: Category Three Test Results (HPL data)

Feature	% Servers Unconditionally Compliant	% Servers Not Compliant
OPTIONS	26.8–32.3	0.8–2.7
TRACE	94.3–97.3	0.2–1.8
FOO	54.5–60.3	5.3–7.1
POST, Expect	54.6–63.2	31.0–32.0
Incorrect URL	75.9–80.5	5.3–8.2
Long URL	62.7	2.0
I-U-S (1123)	40.0–41.7	57.1–59.3
I-U-S (1036)	40.0–41.7	57.1–59.3
I-U-S (ANSI C)	40.0–41.7	57.1–59.3

I-U-S == `If-Unmodified-Since` with `Date` in RFC 1123/1036/ANSI-C formats.
Long URL test done only in June '99.

the servers that do implement this feature correctly understood all of the three required date formats.

7.3 Intersection of three studies

We examined the sites that were in the top 150 in all three study periods. Of the sites claiming to be running HTTP/1.1, 44% of these sites ran Apache, 37% Netscape and 17% IIS. The results were quite consistent with the overall results presented in Section 7. For example, all Apache servers were unconditionally compliant across all Category One tests, as were 71% of IIS (4.0 and 5.0) servers. None of the Netscape servers passed all three Category One tests for unconditional compliance..

7.4 Reasons for non-compliance

There appear to be several reasons for non-compliance on the part of servers. Some of the reasons are subtle and may not be known even to the server implementors. While most instances of a Microsoft-IIS/4.0 site tested yielded the proper `400 Bad Request` to a HTTP/1.1 request without the `Host:` header, 7 of the 174 sites claiming to run IIS/4.0 returned a `200 OK` response. Closer examination and consultation showed that at least one site probably uses an ISAPI filter [Isa] – a dynamically linked library that intercepts requests—with the intention of modifying it before the core server can parse it. In this situation it is the responsibility of the filter to return the appropriate HTTP headers. The failure of many filters to do this correctly suggests that current server implementations may not be meeting the needs of the people who use those servers. Server architectures may need to be redesigned so that the server retains responsibility for handling the HTTP headers while providing users with the functionality they desire. Some sites have purposely configured their servers to return a success response with HTML text indicating that an error has occurred. Even though the server implementation may be compliant, its configuration causes compliance failure, suggesting that the server does not provide its users with desired functionality.

Our results showed that a significant number of sites disabled HTTP/1.1 features such as persistent connections. There are at least three reasons for this behaviour. First, some sites may be concerned about the performance impact of using such features. One server vendor states that "If your site has hits from many users at any time, then persistent connections may not be good for your server." [IBM]. However, no performance evaluation results were provided to substantiate this statement. Second, problems with TCP implementations on some clients has been reported to cause unexpected behaviour in some browsers when persistent connections are used [IBM]. Some sites may therefore choose to disable persistent connections in order to avoid receiving complaints from visitors to their Web site. Third, some HTTP/1.1 servers actually shipped with persistent connections disabled. If the default configuration was used (or if this feature was not specifically enabled) then the site would not support persistent connections. Early versions of a vendor's server [IBM] had persistent connections enabled by default, but some later versions of the had persistent connections disabled by default. Server implementations can thus become less compliant in newer versions.

8 Conclusions and future work

We examined compliance to the HTTP/1.1 protocol of the most popular Web sites in the world. Although many of the popular sites claim to run HTTP/1.1, some even fail the most basic compliance requirements. Many others run with a number of the significant HTTP/1.1 improvements turned off. A presentation by one author of early results of this work [Kri99] led to a prominent browser making persistent connections the default, a prominent server vendor fixing a denial of service attack problem, and another prominent server handling some of the compliance errors. It is clear to us based on our tests and conversations with server developers that MUST level conditions are more likely to be taken seriously. The more damaging scenarios outlined (such as the one that led to server crashes) should be changed from SHOULD to MUST and we have recommended so to the IETF. Including an appendix in the HTTP specification that highlights all of the MUST and SHOULD level conditions may assist implementors in developing compliant products.

We described the first phase of our experiment examining a list of popular servers with a suite of static tests. Ongoing extensions include dynamic tests examining response headers as they arrive and generating subsequent requests based on the response. It may also be useful to examine entire user sessions rather than just single requests for the home page (or text portion thereof). Currently, we use offline analysis to check for protocol compliance. Incorporating this process into the probing mechanism would simplify the conformance checks. A natural extension is testing compliance of proxies. However, testing proxy compliance is significantly harder than testing servers or clients. An HTTP message can go through several proxies, only some of which may be HTTP/1.1 proxies. Compliant HTTP/1.1 proxies can be detected by the presence of `Via` headers but we would not be able to identify the HTTP/1.0 or non-compliant HTTP/1.1 proxies.

Acknowledgment

We thank Anja Feldmann, Roy Fielding, Jim Gettys, Richard Gray, David Kristol, Scott Lawrence, Jeff Mogul, and David Mosberger for answers to various questions. We thank Michel Beaudouin-Lafon, James Griffioen, Eduardo Krell, and Graeme Yates for giving us access to machines. We thank Mediametrix, Netcraft, Hot100 and other sites that periodically run surveys on popular sites and present server penetration statistics. We thank Jennifer Rexford for comments on an earlier draft.

References

[Bra97] S. Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119, IETF, March 1997. `ftp://ftp.ietf.org/rfc2119.txt`.

[ea97] H.F.Nielsen et al. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proc. ACM SIGCOMM*, pages 155–166, August 1997. `http://www.inria.fr/rodeo/sigcomm97/program.html`.

[ea99] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, HTTP Working Group, June 1999. `ftp://ftp.ietf.org/rfc2616.txt`.

[For] HTTP/1.1 feature list report summary. `http://www.w3.org/Protocols/HTTP/Forum/Reports/`.

[For99] 1999 Fortune 500 companies, Fortune volume 139 number 8, April 26 1999.

[Glo98] 1998 Global 500 companies, Fortune Magazine 1998.

[Hot] 100 hot.com. `http://100hot.com/`.

[IBM] HTTP Server for AS/400: Persistent Connections. `http://www.as400.ibm.com/products/http/services/persist.htm`.

[Isa] ISAPI callback functions. `http://support.microsoft.com/support/kb/articles/Q150/3/12.asp`.

[KA99] Balachander Krishnamurthy and Martin Arlitt. PRO-COW: Protocol Compliance on the Web, August 1999. `http://www.research.att.com/~bala/papers/procow-1.ps.gz`.

[KMK99] Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. In *Proc. Eighth International World Wide Web Conference*, Toronto, May 1999.

[Kri99] Balachander Krishnamurthy. PRO-COW: Protocol comliance on the Web, November 1999. Invited plenary session talk at IETF meeting.

[KW00] Balachander Krishnamurthy and Craig E. Wills. Analyzing factors that influence end-to-end web performance. In *Proc. World Wide Web Conference*, May 2000. `http://www.research.att.com/~bala/papers/e2e.ps.gz`.

[Med] Media Metrix.`http://mediametrix.com/`.

[MJ98] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. In *Proceedings of WISP '98, Madison, WI*, pages 59–67, June 1998. `http://www.hpl.hp.com/personal/David_Mosberger/httperf`.

[Net] The Netcraft Web Server Survey. `http://netcraft.co.uk/survey`.

[PM95] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28(1/2):25–35, December 1995.

Nettimer: A Tool for Measuring Bottleneck Link Bandwidth

Kevin Lai Mary Baker
{laik, mgbaker}@cs.stanford.edu
Department of Computer Science, Stanford University

January 29, 2001

Abstract

Measuring the bottleneck link bandwidth along a path is important for understanding the performance of many Internet applications. Existing tools to measure bottleneck bandwidth are relatively slow, can only measure bandwidth in one direction, and/or actively send probe packets. We present the `nettimer` bottleneck link bandwidth measurement tool, the `libdpcap` distributed packet capture library, and experiments quantifying their utility. We test `nettimer` across a variety of bottleneck network technologies ranging from 19.2Kb/s to 100Mb/s, wired and wireless, symmetric and asymmetric bandwidth, across local area and cross-country paths, while using both one and two packet capture hosts. In most cases, `nettimer` has an error of less than 10%, but at worst has an error of 40%, even on cross-country paths of 17 or more hops. It converges within 10KB of the first large packet arrival while consuming less than 7% of the network traffic being measured.

1 Introduction

Network bandwidth continues to be a critical resource in the Internet because of the heterogeneous bandwidths of access technologies and file sizes. This can cause an unaware application to stream a 5GB video file over a 19.2Kb/s cellular data link or send a text-only version of a web site over a 100Mb/s link. Knowledge of the bandwidth along a path allows an application to avoid such mistakes by adapting the size and quality of its content [FGBA96] or by choosing a web server or proxy with higher bandwidth than its replicas [Ste99].

Existing solutions to this problem have examined HTTP throughput [Ste99], TCP throughput [MM96], available bandwidth [CC96a], or bottleneck link bandwidth. Although HTTP and TCP are the current dominant application and transport protocols in the Internet, other applications and transport protocols (e.g. for video and audio streaming) have different performance characteristics. Consequently, their performance cannot be predicted by HTTP and TCP throughput. Available bandwidth (when combined with latency, loss rates, and other metrics) can predict the performance of a wide variety of applications and transport protocols. However, available bandwidth depends on both bottleneck link bandwidth and cross traffic. Cross traffic is highly variable in different places in the Internet and even highly variable in the same place. Developing and verifying the validity of an available bandwidth algorithm that deals with that variability is difficult.

In contrast, bottleneck link bandwidth is well understood in theory [Kes91] [Bol93] [Pax97] [LB00], and techniques to measure it are straightforward to validate in practice (see Section 4). Moreover, bottleneck link bandwidth measurement techniques have been shown to be accurate and fast in simulation [LB99]. Furthermore, in some parts of the Internet, available bandwidth is frequently equal to bottleneck link bandwidth because either bottleneck link bandwidth is small (e.g. wireless, modem, or DSL) or cross traffic is low (e.g. LAN). In addition to bottleneck link bandwidth's current utility, it can help the development of accurate and validated available bandwidth measurement techniques because of available bandwidth's dependence on bottleneck link bandwidth.

However, current tools to measure link bandwidth 1) measure all link bandwidths instead of just the bottleneck, 2) only measure the bandwidth in one direction, and/or 3) actively send probe packets. The tools `pathchar` [Jac97], `clink` [Dow99], `pchar` [Mah00], and `tailgater` [LB00] measure all of the link bandwidths along a path, which can be time-consuming and unnecessary for applications that only want to know the bottleneck bandwidth. Furthermore, these tools and `bprobe` [CC96b] can only measure bandwidth in one direction. These tools, `tcpanaly` [Pax97], and `pathrate`

[DRM01] actively send their own probe traffic, which can be more accurate than passively measuring existing traffic, but also results in higher overhead [LB00]. The `nettimer-sim` [LB99] tool only works in simulation.

Our contributions are the `nettimer` bottleneck link bandwidth measurement tool, the `libdpcap` distributed packet capture library, and experiments quantifying their utility. Unlike current tools, `nettimer` can passively measure the bottleneck link bandwidth along a path in real time. `Nettimer` can measure bandwidth in one direction with one packet capture host and in both directions with two packet capture hosts. In addition, the `libdpcap` distributed packet capture library allows measurement programs like `nettimer` to efficiently capture packets at remote hosts while doing expensive measurement calculations locally. Our experiments indicate that in most cases `nettimer` has less than 10% error whether the bottleneck link technology is 100Mb/s Ethernet, 10Mb/s Ethernet, 11Mb/s WaveLAN, 2Mb/s WaveLAN, ADSL, V.34 modem, or CDMA cellular data. `Nettimer` converges within 10308 bytes of the first large packet arrival. Even when measuring a 100Mb/s bottleneck, `nettimer` only consumes 6.34% of the network traffic being measured, and 4.52% of the cycles on the 366MHz remote packet capture server and 57.6% of the cycles on the 266MHz bandwidth computation machine.

The rest of the paper is organized as follows. In Section 2 we describe the packet pair property of FIFO-queueing networks and show how it can be used to measure bottleneck link bandwidth. In Section 3 we describe how we implement the packet pair techniques described in Section 2, including our distributed packet capture architecture and API. In Section 4, we present preliminary results quantifying the accuracy, robustness, agility, and efficiency of the tool. In Section 6, we conclude.

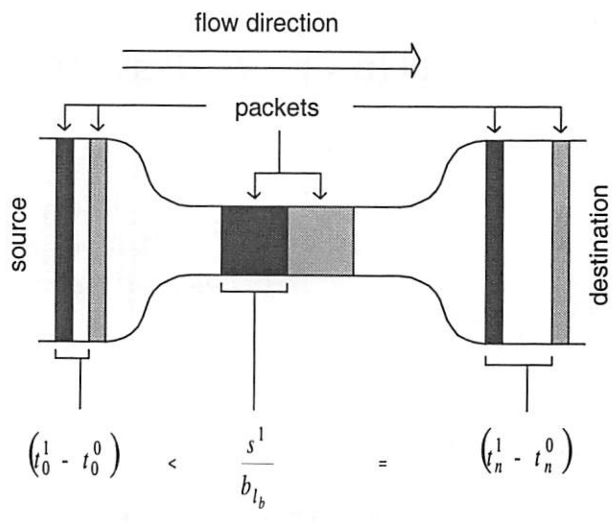
2 Packet Pair Technique

In this section we describe the packet pair property of FIFO-queueing networks and show how it can be used to measure bottleneck link bandwidth.

2.1 Packet Pair Property of FIFO-Queueing networks

The packet pair property of FIFO-queueing networks predicts the difference in arrival times of two packets of the same size traveling from the same source to the same destination:

$$t_n^1 - t_n^0 = \max\left(\frac{s_1}{b_l}, t_0^1 - t_0^0\right) \quad (1)$$



flow direction
packets
source
destination
$\left(t_0^1 - t_0^0\right) < \frac{s^1}{b_{l_b}} = \left(t_n^1 - t_n^0\right)$

Figure 1: This figure shows two packets of the same size traveling from the source to the destination. The wide part of the pipe represents a high bandwidth link while the narrow part represents a low bandwidth link. The spacing between the packets caused by queueing at the bottleneck link remains constant downstream because there is no additional downstream queueing.

where t_n^0 and t_n^1 are the arrival times of the first and second packets respectively at the destination, t_0^0 and t_0^1 are the transmission times of the first and second packets respectively, s_1 is the size of the second packet, and b_l is the bandwidth of the bottleneck link.

The intuitive rationale for this equation (a full proof is given in [LB00]) is that if two packets are sent close enough together in time to cause the packets to queue together at the bottleneck link ($\frac{s_1}{b_l} > t_0^1 - t_0^0$), then the packets will arrive at the destination with the same spacing ($t_n^1 - t_n^0$) as when they exited the bottleneck link ($\frac{s_1}{b_l}$). The spacing will remain the same because the packets are the same size and no link downstream of the bottleneck link has a lower bandwidth than the bottleneck link (as shown in Figure 1, which is a variation of a figure from [Jac88]).

This property makes several assumptions that may not hold in practice. First, it assumes that the two packets queue together at the bottleneck link and at no later link. This could by violated by other packets queueing between the two packets at the bottleneck link, or packets queueing in front of the first, the second or both packets downstream of the bottleneck link. If any of these events occur, then Equation 1 does not hold. In Section 2.2, we describe how to mitigate this limitation by filtering out samples that suffer undesirable queueing.

In addition, the packet pair property assumes that the two packets are sent close enough in time that they queue together at the bottleneck link. This is a problem for very high bandwidth bottleneck links and/or for passive measurement. For example, from Equation 1, to cause queueing between two 1500 byte packets at a 1Gb/s bottleneck link, they would have to be transmitted no more than 12 microseconds apart. An active technique is more likely than a passive one to satisfy this assumption because it can control the size and transmission times of its packets. However, in Section 2.2, we describe how passive techniques can detect this problem and sometimes filter out its effect.

Another assumption of the packet pair property is that the bottleneck router uses FIFO-queueing. If the router uses fair queueing, then packet pair measures the available bandwidth of the bottleneck link [Kes91].

Finally, the packet pair property assumes that transmission delay is proportional to packet size and that routers are store-and-forward. The assumption that transmission delay is proportional to packet size may not be true if, for example, a router manages its buffers in such a way that a 128 byte packet is copied more than proportionally faster than a 129 byte packet. However, this effect is usually small enough to be ignored. The assumption that routers are store-and-forward (they receive the last bit of the packet before forwarding the first bit) is almost always true in the Internet.

Using the packet pair property, we can solve Equation 1 for b_l, the bandwidth of the bottleneck link:

$$b_l = \frac{s_1}{t_n^1 - t_n^0} \tag{2}$$

We call this the *received* bandwidth because it is bandwidth measured at the receiver. When filtering in the next section, we will also use the the bandwidth measured at the sender (the *sent* bandwidth):

$$\frac{s_1}{t_0^1 - t_0^0} \tag{3}$$

2.2 Filtering Techniques

In this section, we describe in more detail how the assumptions in Section 2.1 can be violated in practice and how we can filter out this effect. Using measurements of the sizes and transmission and arrival times of several packets and Equation 1, we can get samples of the received bandwidth. The goal of a filtering technique is to determine which of these samples indicate the bottleneck link bandwidth and which do not. Our approach is to develop a filtering function that gives higher priority to the good samples and lower priority to the bad samples.

Before describing our filtering functions, we differentiate between the kinds of samples we want to keep and those we want to filter out. Figure 2 shows one case that satisfies the assumptions of the packet pair property and three cases that do not. There are other possible scenarios but they are combinations of these cases.

Case A shows the ideal packet pair case: the packets are sent sufficiently quickly to queue at the bottleneck link and there is no queueing after the bottleneck link. In this case the bottleneck bandwidth is equal to the received bandwidth and we do not need to do any filtering.

In case B, one or more packets queue between the first and second packets, causing the second packet to fall farther behind than would have been caused by the bottleneck link. In this case, the received bandwidth is less than the bottleneck bandwidth by some unknown amount, so we should filter this sample out.

In case C, one or more packets queue before the first packet after the bottleneck link, causing the second packet to follow the first packet closer than would have been caused by the bottleneck link. In this case, the received bandwidth is greater than the bottleneck bandwidth by some unknown amount, so we should filter this sample out.

In case D, the sender does not send the two packets close enough together, so they do not queue at the bottleneck link. In this case, the received bandwidth is less than the bottleneck bandwidth by some unknown amount, so we should filter this sample out. Active techniques can avoid case D samples by sending large packets with little spacing between them, but passive techniques are susceptible to them. Examples of case D traffic are TCP acknowledgements, voice over IP traffic, remote terminal protocols like telnet and ssh, and instant messaging protocols.

2.2.1 Filtering using Density Estimation

To filter out the effect of case B and C, we use the insight that samples influenced by cross traffic will tend not to correlate with each other while the case A samples will correlate strongly with each other [Pax97] [CC96b]. This is because we assume that cross traffic will have random packet sizes and will arrive randomly at the links along the path. In addition, we use the insight that packets sent with a low bandwidth that arrive with a high bandwidth are definitely from case C and can be filtered out [Pax97]. Figure 3 shows a hypothetical example of how we apply these insights. Using the second insight, we eliminate the case C samples above the received bandwidth = sent bandwidth ($x = y$) line. Of the remaining samples, we calculate their smoothed distribution and pick the point with the highest density as the bandwidth.

There are many ways to compute the density function of a set of samples [Pax97] [CC96b], including us-

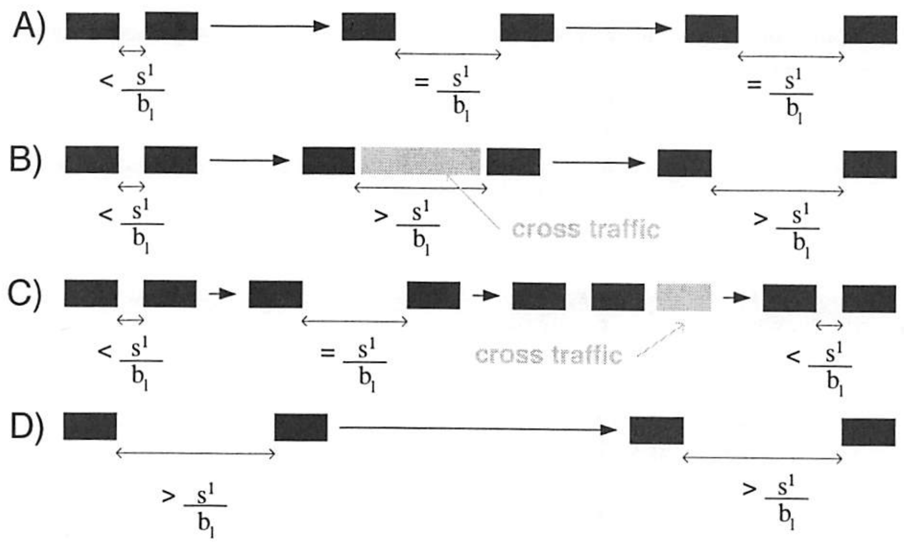


Figure 2: This figure shows four cases of how the spacing between a pair of packets changes as they travel along a path. The black boxes are packets traveling from a source on the left to a destination on the right. Underneath each pair of packets is their spacing relative to the spacing caused by the bottleneck link. They gray boxes indicate cross traffic that causes one or both of the packets to queue.

ing a histogram. However, histograms have the disadvantages of fixed bin widths, fixed bin alignment, and uniform weighting of points within a bin. Fixed bin widths make it difficult to choose an appropriate bin width without previously knowing something about the distribution. For example, if all the samples are around 100,000, it would not be meaningful to choose a bin width of 1,000,000. On the other hand, if all the samples are around 100,000,000, a bin width of 10,000 could flatten an interesting maxima. Another disadvantage is fixed bin alignment. For example, two points could lie very close to each other on either side of a bin boundary and the bin boundary ignores that relationship. Finally, uniform weighting of points within a bin means that points close together will have the same density as points that are at opposite ends of a bin. The advantage of a histogram is its speed in computing results, but we are more interested in accuracy and robustness than in saving CPU cycles.

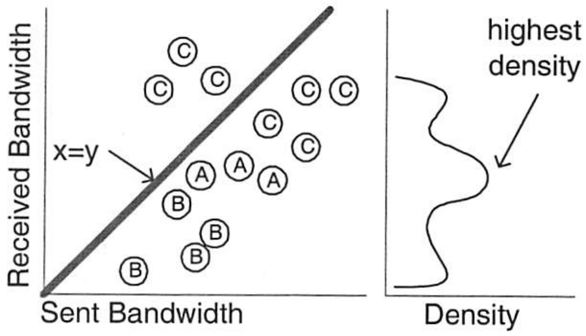


Figure 3: The left graph shows some packet pair samples plotted using their received bandwidth against their sent bandwidth. "A" samples correspond to case A, etc. The right graph shows the distribution of different values of received bandwidth after filtering out the samples above the $x = y$ line. In this example, density estimation indicates the best result.

To avoid these problems, we use *kernel density estimation* [Sco92]. The idea is to define a kernel function $K(t)$ with the property

$$\int_{-\infty}^{+\infty} K(t)dt = 1 \tag{4}$$

Then the density at a received bandwidth sample x is

$$d(x) = \frac{1}{n}\sum_{i=1}^{n} K\left(\frac{x - x_i}{c * x}\right) \tag{5}$$

where c is the kernel width ratio, n is the number of points within $c * x$ of x, and x_i is the ith such point. We use the kernel width ratio to control the smoothness of the density function. Larger values of c give a more accurate result, but are also more computationally expensive. We use a c of 0.10. The kernel function we use is

$$K(t) = \left\{ \begin{array}{ll} 1+t & t \leq 0 \\ 1-t & t > 0 \end{array} \right\} \tag{6}$$

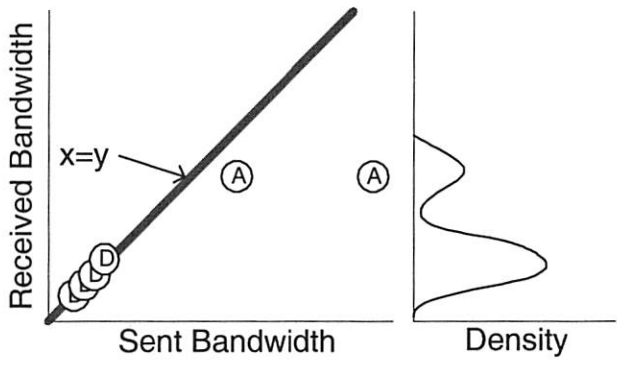


Figure 4: This figure has the same structure as Figure 3. In this example, the ratio of received bandwidth to sent bandwidth is a better indicator than density estimation.

This function gives greater weight to samples close to the point at which we want to estimate density, and it is simple and fast to compute.

2.2.2 Filtering using the Received/Sent Bandwidth Ratio

Although density estimation is the best indicator in many situations, many case D samples can fool density estimation. For example, a host could transfer data in two directions over two different TCP connections to the same correspondent host. If data mainly flows in the forward direction, then the reverse direction would consist of many TCP acknowledgements sent with large spacings and a few data packets sent with small spacings. Figure 4 shows a possible graph of the resulting measurement samples. Density estimation would indicate a bandwidth lower than the correct one because there are so many case D samples resulting from the widely spaced acks.

We can improve the accuracy of our results if we favor samples that show evidence of actually causing queueing at the bottleneck link [LB99]. The case D samples are unlikely to have caused queueing at the bottleneck link because they are so close to the line $x = y$. On the other hand, the case A samples are on average far from the $x = y$ line, meaning that they were sent with a high bandwidth but received with a lower bandwidth. This suggests that they did queue at the bottleneck link.

Using this insight we define the received/sent bandwidth ratio of a received bandwidth sample x to be

$$p(x) = 1 - \frac{ln(x)}{ln(s(x))} \tag{7}$$

where $s(x)$ is the sent bandwidth of x. We take the log of the bandwidths because bandwidths frequently differ by orders of magnitude.

Unfortunately, given two samples with the same sent bandwidth (7) favors the one with the smaller received bandwith. To counteract this, we define the received bandwidth ratio to be

$$r(x) = \frac{ln(x) - ln(x_{min})}{ln(x_{max}) - ln(x_{min})} \tag{8}$$

2.2.3 Composing Filtering Algorithms

We can compose the filtering algorithms described in the previous sections by normalizing their values and taking their linear combination:

$$f(x) = 0.4 * \frac{d(x)}{d(x)_{max}} + 0.3 * p(x) + 0.3 * r(x) \tag{9}$$

where $d(x)_{max}$ is the maximum kernel density value. By choosing the maximum value of $f(x)$ as the bottleneck link bandwidth, we can take into account both the density and the received/sent bandwidth ratio without favoring smaller values of x. The weighting of each of the components is arbitrary. Although it is unlikely that this is the optimal weighting, the results in Section 4.2 indicate that these weightings work well.

2.3 Sample Window

In addition to using a filtering function, we also only use the last w bandwidth samples. This allows us to quickly detect changes in bottleneck link bandwidth (*agility*) while being resistant to cross traffic (*stability*). A large w is more stable than a small w because it will include periods without cross traffic and different sources of cross traffic that are unlikely to correlate with a particular received bandwidth. However, a large w will be less agile than a small w for essentially the same reason. It is not clear to us how to define a w for all situations, so we currently punt on the problem by making it a user-defined parameter in `nettimer`.

3 Implementation

In this section, we describe how `nettimer` implements the algorithms described in the previous section. The issues we address are how to define flows, where to take measurements, and how to distribute measurements.

3.1 Definition of Flows

In Section 2.1, the packet pair property refers to two packets from the same source to the same destination. For `nettimer`, we interpret this flow to be defined by a `(source IP address, destination IP address)` tuple (network level flow), but we could also have interpreted it to be defined by a `(source IP address,` `source port number, destination IP address, source port number)` tuple (transport level flow). The advantage of using transport level flows is that they can penetrate Network Address Translation (NAT) gateways. The advantage of network level flows is that we can aggregate the traffic of multiple transport level flows (e.g. TCP connections) so that we have more samples to work with. We chose network level flows because when we started implementing `nettimer`, NAT gateways were not widespread while popular WWW browsers would open several short TCP connections with servers. We describe a possible solution to this problem in Section 5.

3.2 Measurement Host

In Section 2.1, we assume that we have the transmission and arrival times of packets. In practice, this requires deploying measurement software at both the sender and the receiver, which may be difficult. In this section, we describe how we mitigate this limitation in `nettimer` and the trade-offs of doing so.

3.2.1 Two Hosts

In the ideal case, we can deploy measurement software at both the sender and the receiver. Using this technique, called Receiver Based Packet Pair (RBPP) [Pax97], `nettimer` can employ all of the filtering algorithms described in Section 2.2 because we have both the transmission times and reception times. However, in addition to deploying measurement software at both the sender and the receiver, `nettimer` also needs an architecture to distribute the measurements to interested hosts (described in Section 3.3). We show in Section 4 that RBPP is the most accurate technique.

3.2.2 One Host

When we can only deploy software at one host, we measure the bandwidth from that host to any other host using Sender Based Packet Pair (SBPP) [Pax97] or from any other host to the measurement host using Receiver Only Packet Pair (ROPP) [LB99].

SBPP works by using the arrival times of transport- or application-level acknowledgements instead of the arrival times of the packets themselves. One application of this technique would be to deploy measurement software at a server and measure the bandwidth from the server to clients where software could not be deployed. The issues with this technique are 1) transport- or application-level information, 2) non-per-packet acknowledgements, and 3) susceptibility to reverse path cross traffic. `nettimer` uses transport- or application-level information to match acknowledgements to packets. Currently it only implements this functionality for TCP. Unfortunately, TCP does not have a strict per-packet acknowledgement policy. It only acks every other packet or packets out of order. Furthermore, it sometimes delays acks. Finally, the acks could be delayed by cross traffic on the reverse path, causing more noise for the filtering algorithm to deal with. We show in Section 4 that the non-per-packet acknowledgements make SBPP much less accurate that the other packet pair techniques. We describe a solution to this problem in Section 5.

ROPP works by using only the arrival times of packets. This prevents us from using some of the filtering algorithms described in Section 2.2 because we can no longer calculate the sent bandwidth. One application of this technique would be to deploy measurement software at a client and measure the bandwidth from servers that cannot be modified to the client. We show in Section 4 that in some cases where there is little cross traffic, ROPP is close in accuracy to RBPP.

3.3 Distributed Packet Capture

In this section, we describe our architecture to do distributed packet capture. The `nettimer` tool uses this architecture to measure both transmission and arrival times of packets in the Internet. We first explain our approach and then describe our implementation.

3.3.1 Approach

Our approach is to distinguish between packet capture servers and packet capture clients. The packet capture servers capture packet headers and then distribute them to the clients. The servers do no calculations. The clients receive the packet headers and perform performance calculations and filtering. This allows flexibility in where the packet capture is done and where the calculation is done.

Another possible approach is to do more calculation at the packet capture hosts [MJ98]. The advantage of approach is that packet capture hosts do not have to consume bandwidth by distributing packet headers.

The advantages of separating the packet capture and performance calculation are 1) reducing the CPU burden of the packet capture hosts, 2) gaining more flexibility in the kinds of performance calculations done, and 3) reducing the amount of code that has to run with root privileges. By doing the performance calculation only at the packet capture clients, the servers only capture packets and distribute them to clients. This is especially important if the packet capture server receives packets at a high rate, the packet capture server is collocated with other servers (e.g. a web server), and/or the performance calculation consumes many CPU cycles (as is the case with the filtering algorithm described in Section 2.2). Another advantage is that clients have the flexibility to change their performance calculation code without modifying the packet capture servers. This also avoids the possible security problems of allowing client code to be run on the server. Finally, some operating systems (e.g. Linux) require that packet capture code run with root privileges. By separating the client and server code, only the server runs with root privilege while the client can run as a normal user.

3.3.2 libdpcap

Our implementation of distributed packet capture is the `libdpcap` library. It is built on top of the `libpcap` library [MJ93]. As a result, the `nettimer` tool can measure live in the Internet or from `tcpdump` traces.

To start a `libdpcap` server, the application specifies the parameters `send_thresh`, `send_interval`, `filter_cmd`, and `cap_len`. `send_thresh` is the number of bytes of packet headers the server will buffer before sending them to the client. This should usually be at least the TCP maximum segment size so fewer less than full size packet report packets will sent. `send_interval` is the amount of time to wait before sending the buffered packet headers. This prevents packet headers from languishing at the server waiting for enough data to exceed `send_thresh`. The server sends the buffer when `send_interval` or `send_thresh` is exceeded. The `filter_cmd` specifies which packets should be captured by this server using the `libpcap` filter language. This can cut down on the amount of unnecessary data sent to the clients. For example, to capture only TCP packets between `cs.stanford.edu` and `eecs.harvard.edu` the `filter_cmd` would be "`host cs.stanford.edu and host harvard.stanford.edu and TCP`". `cap_len` specifies how much of each packet to capture.

To start a `libdpcap` client, the application specifies a set of servers to connect to and its own `filter_cmd`. The client sends this `filter_cmd` to the servers with whom it connects. This further restrict the types of packet headers that the client receives.

After a client connects to a server, the server responds with its `cap_len` and its clock resolution. Different machines and operating systems have different clock resolutions for captured packets. For example Linux < 2.2.0 had a resolution of 10ms, while Linux >= 2.2.0 has a resolution < 20 microseconds, almost a thousand times difference. This can make a significant difference in the accuracy of a calculation, so the server reports this clock resolution to the client.

To calculate the bandwidth consumed by the packet reports that the distributed packet capture server sends to its clients, we start with the size of each report: `cap_len` + sizeof(timestamp) (8 bytes) + sizeof(cap_len) (2 bytes) + sizeof(flags) (2 bytes). For TCP traffic, `nettimer` needs at least 40 bytes of packet header. In addition, link level headers consume some variable amount of space. To be safe, we set the capture length to 60 bytes, so each `libdpcap` packet report consumes 72 bytes. 20 of these headers fit in a 1460 byte TCP payload, so the total overhead is approximately 1500 bytes / 20 * 1500 = 5.00%. On a heavily loaded network, this could be a problem. However, if we are only interested in a pre-determined subset of the traffic, we can use the packet filter to reduce the number of packet reports. We experimentally verify this cost in Section 4.2.5 and describe some other ways to reduce it in Section 5.

Table 1: This table shows the different path characteristics used in the experiments. The Short and Long column list the number of hops from host to host for the short and long path respectively. The RTT columns list the round-trip-times of the short and long paths in ms.

Type	Short	RTT	Long	RTT
Ethernet 100 Mb/s	4	1	17	74
Ethernet 10 Mb/s	4	1	17	80
WaveLAN 2 Mb/s	3	4	18	151
WaveLAN 11 Mb/s	3	4	18	151
ADSL	14	19	19	129
V.34 Modem	14	151	18	234
CDMA	14	696	18	727

4 Experiments

In this section we describe the experiments we used to quantify the utility of `nettimer`.

4.1 Methodology

In this section we describe and explain our methodology in running the experiments. Our approach is to take `tcpdump` traces on pairs of machines during a transfer between those machines while varying the bottleneck link bandwidth, path length, and workload. We then run these traces through `nettimer` and analyze the results. Our methodology consists of 1) the network topology, 2) the hardware and software platform, 3) accuracy measurement, 4) the network application workload, and 5) the network environment.

Our network topology consists of a variety of paths (listed in Table 1) where we vary the bottleneck link technology and the length of the path. WaveLAN [wav00] is a wireless local area network technology made by Lucent. ADSL (Asymmetric Digital Subscriber Line) is a high bandwidth technology that uses

Table 2: This table shows the different software versions used in the experiments. The release column gives the RPM package release number.

Name	Version	Release
GNU/Linux Kernel	2.2.16	22
RedHat	7.0	-
tcpdump	3.4	10
tcptrace	5.2.1	1
openssh	2.3.0p1	4
nettimer	2.1.0	1

phone lines to bring connectivity into homes and small businesses. We tested the Pacific Bell/SBC [dsl00] ADSL service. V.34 is an International Telecommunication Union (ITU) [itu00] standard for data communication over analog phone lines. We used the V.34 service of Stanford University. CDMA (Code Division Multiple Access) is a digital cellular technology. We tested CDMA service by Sprint PCS [spr00] with AT&T Global Internet Services as the Internet service provider. These are most of the link technologies that are currently available for users.

In all cases the bottleneck link is the link closest to one of the hosts. This allows us to measure the best and worst cases for `nettimer` as described below. The short paths are representative of local area and metropolitan area networks while the long paths are representative of a cross-country, wide area network. We were not able to get access to an international tracing machine.

All the tracing hosts are Intel Pentiums ranging from 266MHz to 500MHz. The versions of software used are listed in Table 2.

We measure network accuracy by showing a lower bound (TCP throughput on a path with little cross traffic) and an upper bound (the nominal bandwidth specified by the manufacturer). TCP throughput by itself is insufficient because it does not include the bandwidth consumed by link level headers, IP headers, TCP headers and retransmissions. The nominal bandwidth is insufficient because the manufacturer usually measures under conditions that may be difficult to achieve in practice. Another possibility would be for us to measure each of the bottleneck link technologies on an isolated test bed. However, given the number and types of link technologies, this would have been difficult.

The network application workload consists of using `scp` (a secure file transfer program from openssh) to copy a 7476723 byte MP3 file once in each directions along a path. The transfer is terminated after five minutes even if the file has not been fully transferred.

We copy the file in both directions because 1) the ADSL technology is asymmetric and we want to measure both bandwidths and 2) we want to take measurements where the bottleneck link is the first link and the last link. A first link bottleneck link is the worst case for `nettimer` because it provides the most opportunity for cross traffic to interfere with the packet pair property. A last link bottleneck link is the best case for the opposite reason.

We copy a 7476723 byte file as a compromise between having enough samples to work with and not having so many samples that traces are cumbersome to work with. We terminate the tracing after five minutes so that we do not have to wait hours for the file to be transferred across the lower bandwidth links.

The network environment centers around the Stanford University campus but also includes the networks of Pacific Bell, Sprint PCS, Harvard University and the ISPs that connect Stanford and Harvard.

We ran five trials so that we could measure the effect of different levels of cross traffic during different times of day and different days of the week. The traces were started at 18:07 PST 12/01/2000 (Friday), 16:36 PST 12/02/2000 (Saturday), 11:07 PST 12/04/2000 (Monday), 18:39 PST 12/04/2000 (Monday), and 12:00 PST 12/05/2000 (Tuesday). We believe that these traces cover the peak traffic times of the networks that we tested on: commute time (Sprint PCS cellular), weekends and nights (Pacific Bell ADSL, Stanford V.34, Stanford residential network), work hours (Stanford and Harvard Computer Science Department networks).

Within the limits of our resources, we have selected as many different values for our experimental parameters as possible to capture some of the heterogeneity of the Internet.

4.2 Results

In this section, we analyze the results of the experiments.

4.2.1 Varied Bottleneck Link

One goal of this work is to determine whether `nettimer` can measure across a wide variety of network technologies. Dealing with different network technologies is not just a matter of dealing with different bandwidths because different technologies have very different link and physical layer protocols that could affect bandwidth measurement.

Using Table 3, we examine the short path Receiver Based Packet Pair results for the different technologies. This table gives the mean result over all the times and days of the TCP throughput and Receiver Based result reported by `nettimer`.

The Ethernet 100Mb/s case and to a lesser extent the Ethernet 10Mb/s case show that using TCP to measure the bandwidth of a high bandwidth link can be

Table 3: This table summarizes `nettimer` results over all the times and days. "Type" lists the different bottleneck technologies. "D" lists the direction of the transfer. "u" and "d" indicate that data is flowing away from or towards the bottleneck end, respectively. "Path" indicates whether the (l)ong or (s)hort path is used. "N" lists the nominal bandwidth of the technology. "TCP" lists the TCP throughput. "RB" lists the `nettimer` results for Receiver Based packet pair. (σ) lists the standard deviation over the different traces.

High bandwidth technologies (Mb/s):

Type	D	P	N	TCP (σ)	RB (σ)
Ethernet	d	s	100	21.22 (.13)	88.39 (.01)
Ethernet	d	l	100	2.09 (.41)	59.15 (.04)
Ethernet	u	s	100	19.92 (.05)	90.16 (.06)
Ethernet	u	l	100	1.51 (.58)	92.03 (.02)
Ethernet	d	s	10.0	6.56 (.06)	9.65 (.00)
Ethernet	d	l	10.0	1.85 (.14)	9.62 (.00)
Ethernet	u	s	10.0	7.80 (.03)	9.46 (.00)
Ethernet	u	l	10.0	1.66 (.21)	9.30 (.02)
WaveLAN	d	s	11.0	4.33 (.16)	6.52 (.20)
WaveLAN	d	l	11.0	1.63 (.13)	7.25 (.22)
WaveLAN	u	s	11.0	4.64 (.17)	5.30 (.12)
WaveLAN	u	l	11.0	1.51 (.32)	5.07 (.14)
WaveLAN	d	s	2.0	1.38 (.01)	1.48 (.02)
WaveLAN	d	l	2.0	1.05 (.09)	1.47 (.02)
WaveLAN	u	s	2.0	1.07 (.05)	1.21 (.01)
WaveLAN	u	l	2.0	0.87 (.26)	1.17 (.00)
ADSL	d	s	1.5	1.21 (.01)	1.24 (.00)
ADSL	d	l	1.5	1.16 (.01)	1.23 (.00)

Low bandwidth technologies (Kb/s):

Type	D	P	N	TCP (σ)	RB (σ)
ADSL	u	s	128	96.87 (.19)	109.28 (.00)
ADSL	u	l	128	107.0 (.01)	109.51 (.00)
V.34	d	s	33.6	26.43 (.04)	27.04 (.03)
V.34	d	l	33.6	26.77 (.04)	27.52 (.04)
V.34	u	s	33.6	27.98 (.01)	28.62 (.01)
V.34	u	l	33.6	28.05 (.00)	28.82 (.00)
CDMA	d	s	19.2	5.30 (.05)	10.88 (.05)
CDMA	d	l	19.2	5.15 (.09)	10.83 (.09)
CDMA	u	s	19.2	6.76 (.24)	18.48 (.05)
CDMA	u	l	19.2	6.50 (.53)	17.21 (.11)

inaccurate and/or expensive. For both Ethernets, the TCP throughput is significantly less than the nominal bandwidth. This could be caused by cross traffic, not being able to open the TCP window enough, bottlenecks in the disk, inefficiencies in the operating system, and/or the encryption used by the `scp` application. In general, using TCP to measure bandwidth requires actually filling that bandwidth. This may be expensive in resources and/or inaccurate. We have no explanation for the RBPP result of 59Mb/s for the down long path.

In the WaveLAN cases, both the `nettimer` estimate and the TCP throughput estimate deviate significantly from the nominal. However, another study [BPSK96] reports a peak TCP throughput over WaveLAN 2Mb/s of 1.39Mb/s. We took the traces with a distance of less than 3m between the wireless node and the base station and there were no other obvious sources of electromagnetic radiation nearby. We speculate that the 2Mb/s and 11Mb/s nominal rates were achieved in an optimal environment shielded from external radio interference and conclude that the `nettimer` reported rate is close to the actual rate achievable in practice.

Another anomaly is that the `nettimer` measured WaveLAN bandwidths are consistently higher in the down direction than in the up direction. This is unlikely to be `nettimer` calculation error because the TCP throughputs are similarly asymmetric. Since the hardware in the PCMCIA NICs used in the host and the base station are identical, this is most likely due to an asymmetry in the MAC-layer protocol.

The `nettimer` measured ADSL bandwidth consistently deviates from the nominal by 15%-17%. Since the TCP throughput is very close to the `nettimer` measured bandwidth, this deviation is most likely due to the overhead from PPP headers and byte-stuffing (Pacific Bell/SBC ADSL uses PPP over Ethernet) and the overhead of encapsulating PPP packets in ATM (Pacific Bell/SBC ADSL modems use ATM to communicate with their switch). Link layer overhead is also the likely cause of the deviation in V.34 results.

The CDMA results exhibit an asymmetry similar to the WaveLAN results. However, we are fairly certain that the base station hardware is different from our client transceiver and this may explain the difference. However, this may also be due to an interference source close to the client and hidden from the base station. In addition, since the TCP throughputs are far from both the nominal and the `nettimer` measured bandwidth, the deviation may be due to `nettimer` measurement error.

We conclude that `nettimer` was able to measure the bottleneck link bandwidth of the different link technologies with a maximum error of 41%, but in most cases with an error less than 10%.

4.2.2 Resistance to Cross Traffic

We would expect that the long paths would have more cross traffic than the short paths and therefore interfere with `nettimer`. In addition, we would expect that bandwidth in the up direction would be more difficult to measure than bandwidth in the down direction because packets have to travel the entire path before their arrival time can be measured.

However, Table 3 shows that the RBPP technique and `nettimer`'s filtering algorithm are able to filter out the effect of cross traffic such that `nettimer` is accurate for long paths even in the up direction.

In contrast, ROPP is much less accurate on the up paths than on the down paths (Section 4.2.3).

It was pointed out by an anonymous reviewer that there may be environments (e.g. a busy web server) where packet sizes and arrival times are highly correlated, which would violate some of the assumptions described in Section 2.2.1. There are definitely parts of the Internet containing technologies and/or traffic patterns so different from those described here that they cause `nettimer`'s filtering algorithm to fail. One example is multi-channel ISDN, which is no longer in common use in the United States. We simply claim that `nettimer` is accurate in a variety of common cases which justifies further investigation into its effectiveness in other cases.

4.2.3 Different Packet Pair Techniques

In this section, we examine the relative accuracy of the different packet pair techniques. Table 4 shows the Receiver-Only and Sender-Based results of one day's traces.

Sender Based Packet Pair is not particularly accurate, reporting 20%-50% of the estimated bandwidth, even on the short paths. As mentioned before, this is most likely the result of passively using TCP's non-per-packet acknowledgements and delayed acknowledgements. We discuss possible solutions to this in Section 5.

In the down direction for both long and short paths, Receiver Only Packet Pair is almost as accurate as RBPP. In contrast, Receiver Only Packet Pair is amazingly inaccurate in the up direction. For ROPP to make an accurate measurement, packets have to preserve their spacing resulting from the first link during their journey along all of the later links. ROPP cannot filter using the sent bandwidth (Section 2.2.2) because it does not have the cooperation of the sending host. Consequently, ROPP has poor accuracy compared to RBPP.

Table 4: This table shows 11:07 PST 12/04/2000 `nettimer` results. "Type" lists the different bottleneck technologies. "D" lists the direction of the transfer. "u" and "d" indicate that data is flowing away from or towards the bottleneck end, respectively. "P" indicates whether the (l)ong or (s)hort path is used. "Nom" lists the nominal bandwidth of the technology. "RO" and "SB" list the Receiver Only or Sender Based packet pair bandwidths respectively. (σ) lists the standard deviation over the duration of the connection.

High bandwidth technologies (Mb/s):

Type	D	P	Nom	RO (σ)	SB (σ)
Ethernet	d	s	100.0	87.69 (.12)	29.22 (.46)
Ethernet	d	l	100.0	63.65 (.27)	22.56 (1.8)
Ethernet	u	s	100.0	697.39 (.12)	52.28 (.22)
Ethernet	u	l	100.0	706.34 (.04)	13.96 (1.1)
Ethernet	d	s	10.0	9.65 (.03)	92.80 (.49)
Ethernet	d	l	10.0	9.65 (.04)	12.44 (2.5)
Ethernet	u	s	10.0	84.03 (.47)	4.63 (.04)
Ethernet	u	l	10.0	97.85 (.04)	6.42 (2.6)
WaveLAN	d	s	11.0	8.00 (.22)	3.40 (3.1)
WaveLAN	d	l	11.0	8.36 (.25)	2.11 (.33)
WaveLAN	u	s	11.0	11.30 (.03)	2.43 (.29)
WaveLAN	u	l	11.0	11.56 (.03)	1.77 (.31)
WaveLAN	d	s	2.0	1.46 (.03)	0.76 (.03)
WaveLAN	d	l	2.0	1.46 (.04)	0.74 (.05)
WaveLAN	u	s	2.0	1.20 (.03)	0.60 (.00)
WaveLAN	u	l	2.0	1.20 (.03)	0.59 (.06)
ADSL	d	s	1.5	1.24 (.03)	0.59 (.04)
ADSL	d	l	1.5	1.24 (.04)	0.59 (.05)

Low bandwidth technologies (Kb/s):

Type	D	P	Nom	RO (σ)	SB (σ)
ADSL	u	s	128.0	465.34 (.04)	54.53 (.01)
ADSL	u	l	128.0	390.58 (.07)	53.89 (.04)
V.34	d	s	33.6	26.43 (.04)	6.94 (.95)
V.34	d	l	33.6	28.54 (.07)	5.35 (1.3)
V.34	u	s	33.6	831.67 (3.6)	14.45 (.05)
V.34	u	l	33.6	674.15 (2.7)	14.50 (.03)
CDMA	d	s	19.2	11.40 (.17)	9.85 (.36)
CDMA	d	l	19.2	12.07 (.09)	12.45 (.36)
CDMA	u	s	19.2	508.12 (1.5)	11.08 (.26)
CDMA	u	l	19.2	484.07 (1.2)	7.56 (2.0)

4.2.4 Agility

In this section, we examine how quickly `nettimer` calculates bandwidth when a connection starts. Figure 5 shows the bandwidth that `nettimer` using RBPP reports at the beginning of a connection. The connection begins 1.88 seconds before the first point on the graph. `nettimer` initially reports a low bandwidth, then a (correct) high bandwidth, then a low bandwidth, then converges at the high bandwidth. The total time from the beginning of the connection to convergence is 3.72 seconds. It takes this long because `scp` requires several round trips to authenticate and negotiate the encryption.

If we measure from when the data packets begin to flow, `nettimer` converges when the 8th data packet arrives, 8.4 ms after the first data packet arrives, 10308 bytes into the connection. TCP would have reported the throughput at this point as 22.2Kb/s. Converging within 10308 bytes means that an adaptive web server could measure bandwidth using just the text portion of most web pages and then adapt its images based on that measurement.

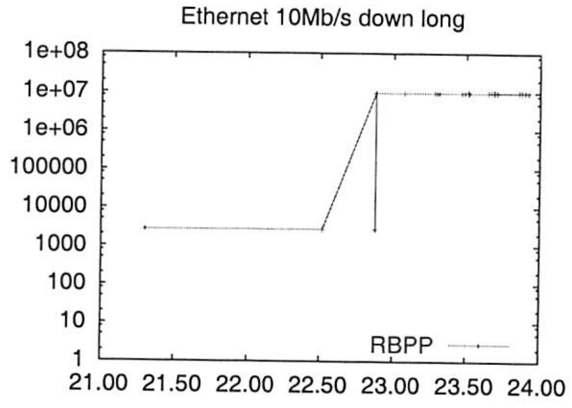


Figure 5: This graph shows the bandwidth reported by `nettimer` using RBPP at a particular time for Ethernet 10Mb/s in the down direction along the long path. The Y-axis shows the bandwidth in b/s on a log scale. The X-axis shows the number of seconds since tracing began.

Table 5: This table shows the CPU overhead consumed by nettimer and the application it is measuring. "User" lists the user-level CPU seconds consumed. "System" lists the system CPU seconds consumed. "Elapsed" lists the elapsed time that the program was running. "% CPU" lists (User + System) / `scp` Elapsed time.

Name	User	System	Elapsed	% CPU
server	.31	.43	32.47	4.52%
client	9.28	.15	26.00	57.6%
`scp`	.050	.21	16.37	1.59%

4.2.5 Resources Consumed

In this section, we quantify the resources consumed by `nettimer`. In contrast to the other experiments where we took traces and then used `nettimer` to process the traces, in this experiment, `nettimer` captured its own packets and calculated the bandwidth as the connection was in progress. We measure the Ethernet 100Mb/s short up path because this requires the most efficient processing. We use `scp` and copy the same file as before. The distributed packet capture server ran on an otherwise unloaded 366MHz Pentium II while the packet capture client and `nettimer` processing ran on an otherwise unloaded 266MHz Pentium II.

Table 5 lists the CPU resources consumed by each of the components. The CPU cycles consumed by the distributed packet capture server are negligible, even for a 366MHz processor on a 100Mb/s link. `Nettimer` itself does consume a substantial number of CPU seconds to classify packets into flows and run the filtering algorithm. However, this was on a relatively old 266MHz machine and this functionality does not need to be co-located with the machine providing the actual service being measured (in this case the `scp` program).

Transferring the packet headers from the `libdpcap` server to the client consumed 473926 bytes. Given that the file transferred is 7476723 bytes, the overhead is 6.34%. This is higher than the 5.00% predicted in Section 3.3.2 because 1) `scp` transfers some extra data for connection setup, 2) some data packets are retransmitted, and most significantly, 3) the `libdpcap` server captures its own traffic. The server captures its own traffic because it does not distinguish between the `scp` data packets and its own packet header traffic, so it captures the headers of packets containing the headers of packets containing headers and so on. Fortunately, there is a limit to the recursion so the net overhead is close to the predicted overhead.

5 Future Work

In this section, we describe some possible future improvements to the `nettimer` implementation. One improvement would be to determine what the optimal weighting of components in the filtering algorithm (Section 2.2.3) is. Another improvement would be to allow runtime choice of flow definition (Section 3.1), so that we could measure bandwidth behind NAT gateways. Another improvement would be to allow distributed packet capture servers to randomly sample traffic to reduce the amount of bandwidth that packet reports consume. Finally, we could add an active probing component like [Sav99] which can cause large packets to flow in both directions from hosts without special measurement software and can cause prompt acknowledgements to flow back to the sender. The `pathrate` [DRM01] tool shows that active packet pair can be very accurate.

6 Conclusion

In this paper, we describe the trade-offs involved in implementing `nettimer`, a Packet Pair-based tool for passively measuring bottleneck link bandwidths in real time in the Internet. We show its utility across a wide variety of bottleneck link technologies ranging from 19.2Kb/s to 100Mb/s, wired and wireless, symmetric and asymmetric bandwidth, across local area and cross country paths, while using both one and two packet capture hosts.

In the future, we hope that `nettimer` will ease the creation of adaptive applications, provide more insight for network performance analysis, and lead to the development of more precise performance measurement algorithms.

7 Acknowledgments

We would like to thank several people without whom this research would have been extremely difficult. The anonymous USITS reviewers provided many valuable comments. Margo Seltzer and Dave Sullivan at Harvard University graciously allowed us to do the critical cross-country tracing at one of their machines. Brian Roberts gave us the last available 100Mb/s Ethernet switch port on our floor. Ed Swierk hosted one of our laptops in his campus apartment. T.J. Giuli lent us his laptop for several weeks.

This work was supported by a gift from NTT Mobile Communications Network, Inc. (NTT DoCoMo). In addition, Kevin Lai was supported in part by a USENIX Scholar Fellowship.

References

[Bol93] Jean-Chrysostome Bolot. End-to-End Packet Delay and Loss Behavior in the Internet. In *Proceedings of ACM SIGCOMM*, 1993.

[BPSK96] Hari Balakrishnan, Venkata Padmanabhan, Srinivasan Seshan, and Randy Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *Proceedings of ACM SIGCOMM*, 1996.

[CC96a] Robert L. Carter and Mark E. Crovella. Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks. Technical Report BU-CS-96-007, Boston University, 1996.

[CC96b] Robert L. Carter and Mark E. Crovella. Measuring Bottleneck Link Speed in Packet-Switched Networks. Technical Report BU-CS-96-006, Boston University, 1996.

[Dow99] Allen B. Downey. Using pathchar to Estimate Internet Link Characteristics. In *Proceedings of ACM SIGCOMM*, 1999.

[DRM01] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. What do packet dispersion techniques measure? In *Proceedings of IEEE INFOCOM*, April 2001.

[dsl00] DSL. http://www.pacbell.com/DSL, 2000.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[itu00] ITU. http://www.itu.int, 2000.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM*, 1988.

[Jac97] Van Jacobson. pathchar. ftp://ftp.ee.lbl.gov/pathchar/, 1997.

[Kes91] Srinivasan Keshav. A Control-Theoretic Approach to Flow Control. In *Proceedings of ACM SIGCOMM*, 1991.

[LB99] Kevin Lai and Mary Baker. Measuring Bandwidth. In *Proceedings of IEEE INFOCOM*, March 1999.

[LB00] Kevin Lai and Mary Baker. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *Proceedings of ACM SIGCOMM*, August 2000.

[Mah00] Bruce A. Mah. pchar. http://www.ca.sandia.gov/~bmah/Software/pchar/, 2000.

[MJ93] Steve McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, 1993.

[MJ98] G. Robert Malan and Farnam Jahanian. An Extensible Probe Architecture for Network Protocol Performance Measurement. In *Proceedings of ACM SIGCOMM*, 1998.

[MM96] M. Mathis and J. Mahdavi. Diagnosing Internet Congestion with a Transport Layer Performance Tool. In *Proceedings of INET*, 1996.

[Pax97] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamics.* PhD thesis, University of California, Berkeley, April 1997.

[Sav99] Stefan Savage. Sting: a TCP-based Network Measurement Tool. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1999.

[Sco92] Dave Scott. *Multivariate Density Estimation: Theory, Practice and Visualization.* Addison Wesley, 1992.

[spr00] Sprint PCS. http://www.sprintpcs.com/, 2000.

[Ste99] Mark R. Stemm. *An Network Measurement Architecture for Adaptive Applications.* PhD thesis, University of California, Berkeley, 1999.

[wav00] WaveLAN. http://www.wavelan.com/, 2000.

CANS: Composable, Adaptive Network Services Infrastructure

Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{xiaodong,weisong,akkerman,vijayk}@cs.nyu.edu

Abstract

Ubiquitous access to sophisticated internet services from diverse end devices across heterogeneous networks requires the injection of additional functionality into the network to handle protocol conversion, data transcoding, and in general bridge disparate network portions. Several researchers have proposed infrastructures for injecting such functionality; however, many challenges remain before these can be widely deployed.

CANS is an application-level infrastructure for injecting application-specific components into the network that focuses on three such challenges: (a) efficient and dynamic composition of individual components; (b) distributed adaptation of injected components in response to system conditions; and (c) support for legacy applications and services. The CANS network view comprises *applications*, stateful *services*, and *data paths* built from mobile soft-state objects called *drivers*. Both services and data paths can be dynamically created and reconfigured: a planning and event propagation model assists in distributed adaptation, and a flexible type-based composition model dictates how new services and drivers are integrated with existing components. Legacy components plug into CANS using an interception layer that virtualizes network bindings and a delegation model.

This paper describes the CANS architecture, and a case study involving a shrink-wrapped client application in a dynamically changing network environment where CANS improves overall user experience.

1 Introduction

The emergence of new networking technologies such as broadband to the home, Wireless 3G [18], and Bluetooth [10], coupled with increasing numbers of network-capable end devices holds the potential for future application services that significantly enhance user experience by providing seamless, ubiquitous access. To take an example, consider the following scenario. Alice, a telecommuting employee, starts her day by initiating a teleconference on her laptop connected to the internet using a wired LAN. During the conference, a hub failure renders the wired LAN unavailable. Fortunately, the service detects this, and seamlessly switches data transmission to a local wireless network while simultaneously degrading picture quality upon recognizing that the wireless LAN has insufficient bandwidth for continuous video at the original resolution and rate. Shortly after, Alice leaves her office to meet a client. She shuts down her laptop, and resumes the teleconference in her car using a PDA connected to a metro-area wireless network. The service further downgrades the media stream (say to only include audio), while recording the full stream at a server that Alice can check offline.

Although the above scenario is compelling, its requirements — rapid creation and deployment of new services, application-aware computation in the network, and dynamic and distributed adaptation — are poorly handled by current internet infrastructure. Moreover, the existing view which hides network characteristics from the application and treats services as standalone entities is incompatible with the large variation in network and end-device characteristics. Current-day data paths can include links with very different bandwidth, delay, and error characteristics, ranging from serial links to wireless to broadband to fiber. Hiding these differences from the application will result in unsatisfactory application performance, and the alternative of providing differentiated service for different networks/end-devices cannot adequately cope with dynamically changing environments.

One solution to these problems is to inject additional functionality into the network that can dynamically adapt to resource characteristics of end-devices and network links by handling activities such as protocol conversion, data transcoding, etc. Several researchers have proposed infrastructures for achieving this goal, ranging from end-point solutions [12, 15] to more distributed alternatives that introduce application-aware functionality either at the network level [17, 3, 20] or at the application level [1, 6, 8]. Although these systems have articulated a common set of high-level architectural requirements, many challenges, particularly with respect to dynamic services management and composition, remain before the infrastructures see widespread deployment.

This paper describes **C**omposable **A**daptive **N**etwork **S**ervices (CANS), an application-level infrastructure for customizing the data path between client applications and services, which focuses on three such challenges:

- *Efficient and Dynamic Composition*, enabling separately defined components to be dynamically instantiated and interconnected using efficient mechanisms (e.g., shared memory within a host).
- *Dynamic and Distributed Adaptation*, enabling adaptation to environment changes along the entire data path while incurring low overhead and maintaining overall application semantics.
- *Support for Legacy Applications and Services*, enabling the latter to be integrated into CANS with minimal effort. Requiring rewrites of each application and service is neither practical nor desirable.

CANS addresses these challenges by constructing networks that include *applications*, stateful *services*, and *data paths* between them built up from mobile soft-state objects called *drivers*. Drivers implement a standard interface, permitting efficient composition and semantics-preserving adaptation. Both services and data paths can be dynamically created and reconfigured: a planning and event propagation facility enables distributed adaptation, and a flexible type-based composition model dictates how new services and drivers are integrated with existing ones. CANS provides three adaptation modes to permit cost-functionality tradeoffs: intra-component, by reconfiguring data paths, and by creating new services and data paths. Legacy components plug into CANS using delegation and an interception layer that transparently virtualizes network bindings, currently TCP sockets.

CANS has been implemented on Windows 2000 clients and Java/RMI-capable intermediate hosts. Each node runs the CANS execution environment, which supports dynamic creation, migration, and adaptation of drivers and services. Experience with a case study involving a shrink-wrapped application (Windows MediaPlayer) in a dynamically changing network environment indicates the potential of our approach: CANS permits dynamic deployment and distributed adaptation of application-aware components to improve overall user experience.

The rest of this paper is organized as follows. Section 2 presents the CANS architecture, with details about its components and distributed adaptation support appearing in Sections 3 and 4. Section 5 presents the CANS implementation and the MediaPlayer case study. Section 6 discusses related efforts, and Section 7 concludes.

2 CANS Architecture

2.1 The Logical View

CANS views networks as consisting of *applications*, *services*, and *data paths* connecting the two. CANS extends the notion of a data path, traditionally limited to data transmission between end points, to include application-specific components dynamically injected by end services, applications, or other entities; these components adapt the data path to physical link characteristics of the underlying network and properties of end devices (see Figure 1(a)).

Components are self-contained pieces of code that can perform a particular activity, e.g., protocol conversion or data transcoding. Components operate on *typed* data streams and are connected with each other based upon compatibility of output and input types (see Section 3 for details). Injected components come in two flavors: stateful *services* and mobile soft-state objects called *drivers*. Services extend the original data path to multiple hops, and drivers generalize the traditional notion of a data path to include data transformation in addition to transmission. The primary reason for distinguishing between drivers and services is to ensure efficiency.

CANS data paths are created dynamically, using information about user preferences, properties of services and client applications, as well as characteristics of the underlying platform. The components which constitute a data path, the interconnections amongst them, and their internal configuration parameters can all be modified at run time. Modifications are triggered based on either system events (e.g., breaking of a network link) or component-initiated events. The CANS infrastructure provides support to efficiently reconfigure data paths, while preserving application semantics.

2.2 The Physical View

The CANS network is realized by partitioning the services and data paths onto physical hosts, connected using existing communication mechanisms. The CANS Execution Environment (EE) serves as the basic runtime environment on these hosts and includes the following functional modules (see Figure 1(b)): *class manager*, *plan manager*, *driver and service manager*, *event manager*, and *resource monitor*.

The class manager handles downloading of component code and instantiation of the components. The plan manager is responsible both for creating the initial plan comprising drivers, services, and data paths in response to a request trapped by the interception layer, as well as replanning in response to system conditions. The driver and service manager maintains information about deployed drivers and manages data path operations, includ-

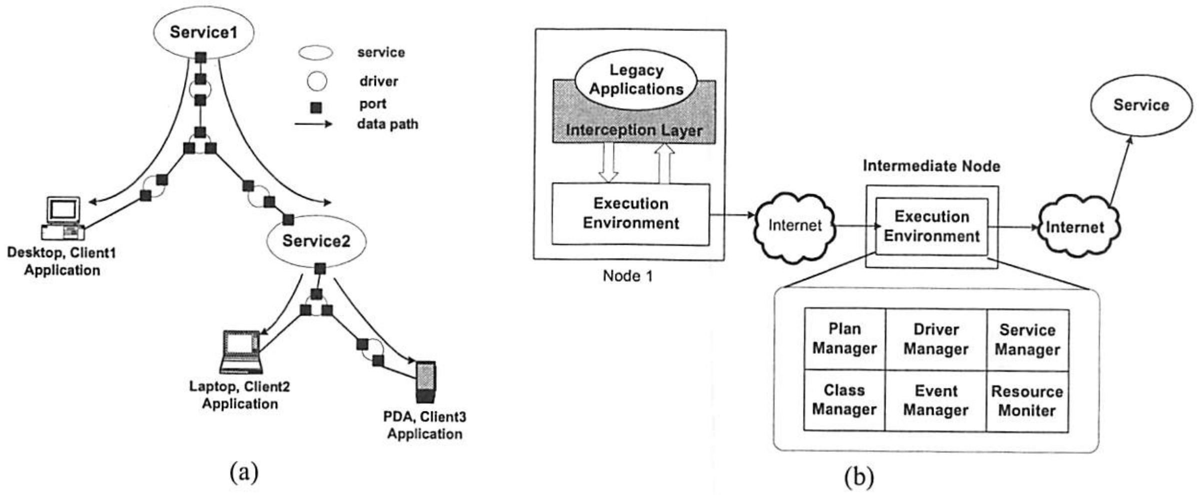


Figure 1. (a) Logical organization of CANS, (b) Physical realization of CANS data paths.

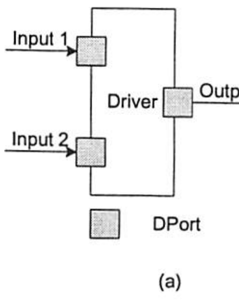
ing inserting new drivers, creating new services, and reconfiguring existing paths as required. The event manager is responsible for receiving both system-level and component-level events and propagating these on to interested components. The resource monitor monitors system conditions such as CPU availability or network interface state, informing the event manager when registered trigger conditions fire.

3 CANS Components

CANS components include drivers, services, and auxillary components that interconnect execution environments, applications, and legacy services.

3.1 Drivers

Drivers serve as the basic building block for constructing adaptation-capable, customized data paths. Drivers are standalone mobile code modules that perform some operation on the data stream. However, to permit their efficient composition and dynamic low-overhead reconfiguration of data paths, drivers are required to adhere to a restricted interface as shown in Figure 2. Specifically,



```
class Driver {
  String driverID;

  TList outTypes(TList inTypes);
  DPortList getPorts();
  DPort getPort(String PortId);
  void push(DInPort input);
  void pull(DOutPort out);
  void raiseEvent(CANSEvent e);
  void registerListener();
}
```
(b)

Figure 2. Driver functionality (a) and interface (b).

1. Drivers consume and produce data using a standard *data port* interface, called a `DPort`. `DPorts` are typed (details below) and distinguished based on whether they are being used for input or output.

2. Drivers are *passive*, moving data from input ports to output ports in a purely demand-driven fashion. Driver activity is triggered only when one of its output ports is checked for data, or one of its input ports receives data.

3. Drivers consume and produce data at the granularity of an integral number of application-specific units, called *semantic segments*. These segments are naturally defined based on the application, e.g., an HTML page or an MPEG frame. Informally, this requirement ensures that the data in an input semantic segment can only influence data in a fixed number of output segments, permitting construction of data path reconfiguration and error recovery strategies that rely upon retransmission at the granularity of semantic segments (see Section 4.2). Note that this property only refers to the logical view of the driver, and admits physical realizations that transmit data at any convenient granularity as long as segment boundaries are somehow demarcated (e.g., with marker messages).

4. Drivers contain only *soft state*, which can be reconstructed simply by restarting the driver. Stated differently, given a semantically equivalent sequence of input segments, a soft-state driver always produces a semantically equivalent sequence of output segments. For example, a Zip driver that produces compressed data will produce semantically equivalent output (i.e., uncompressed to the same string) if presented with the same input strings.

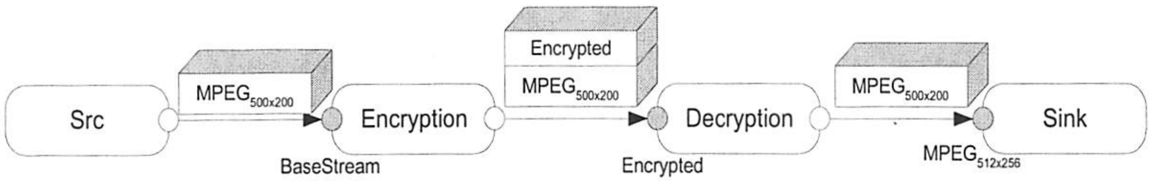


Figure 3. A simple example of type compatibility.

The first two properties enable dynamic composition and efficient transfer of data segments between multiple drivers that are mapped to the same physical host (e.g., via shared memory). Moreover, they permit driver execution to be orchestrated for optimal performance. For example, a single thread can be employed to execute, in turn, multiple driver operations on a single data segment. This achieves nearly the same efficiency, modulo indirect function call overheads, as if driver operations were statically combined into a single procedure call.

The semantic segments and soft-state properties enable low-overhead dynamic adaptation, either within a single driver or across data path segments while preserving application semantics. The driver interface (see Figure 2) permits a driver to create and listen to events, facilitating its participation in distributed adaptation activities.

Type-based Composition

The composability of CANS components (both drivers and services) is decided by compatibility of the type information associated with the input and output ports being connected. The types used in CANS integrate two closely related concepts: *data types* and *stream types*.

CANS data types are the basic unit of type information, represented by a type object that in addition to a unique type name can contain arbitrary attributes and operations for checking type compatibility. Traditional mechanisms such as type hierarchies can still be used to organize data types; however, our scheme permits flexible type compatibility relationships not easily expressibly just by matching type names. For instance, it is possible to define a CANS type for MPEG data, which contains attributes for defining the frame size. An `MPEG` type can be defined compatible with another `MPEG` type as long as the former's frame size is smaller than the latter's, naturally capturing the behavior that a lower resolution MPEG stream can be played on a client platform capable of displaying a higher resolution stream.

CANS stream types capture the aggregate effect of multiple CANS drivers operating upon a typed data stream. Stream types are constructed at run time, and representable as a *stack* of data types. Operations allowed on stream types include *push, pop, peek,* and *clone*, which have the standard meanings.

Each CANS component with m input ports and n output ports defines a function, which maps its input stream types into output stream types:

$$f(T_{in_1}, T_{in_2}, ..., T_{in_m}) \rightarrow (T_{out_1}, T_{out_2}, ..., T_{out_n})$$

where T_{in_i} is the required stream type set for the ith input port, and T_{out_j} is the resulting stream type produced on the jth output port. The type compatibility between an input and an output port, which determines whether two components can be connected, is determined by checking the top of the output port's stream type against the required data type of the input port. Stream type information flows downstream automatically when two ports get connected at run time.

Figure 3 shows an example of the type compatibility scheme. The source produces MPEG data at resolution 500×200, which needs to be supplied to the sink that can consume MPEG data at resolution 512×256 after going through two components that respectively encrypt and decrypt the data. The figure shows the data types on each of the ports as well as the stream types on the connections. To consider an example, the *Encryption* driver accepts data type `BaseStream` and pushes an `Encrypted` type object onto the incoming stream type. The output port of *Src* is compatible with the input port of *Encryption* because the `MPEG` type object extends the `BaseStream` type. Similarly, the output port of *Decryption*, whose affect is to pop the `Encrypted` type from its incoming stream type, is compatible with the input port of *Sink* because of a type-specific compatibility operator for the `MPEG` type that looks at the resolution attributes.

Figure 3 also highlights the composition advantages of representing stream types as a stack of data types. If components were just modeled as consuming data of a particular type and producing data of another type, it would be difficult to express the behavior of the *Encryption* and *Decryption* drivers in a way that permits their use for a variety of generic stream types *without* losing information about the original stream type at the output of the *Decryption* driver. Thus, determining whether the *Decryption* driver's output port is compatible with the input port on *Sink* would require examining the entire data path. In contrast, our stream type representation permits local decision making, a prerequisite for run-time adaptation via dynamic component composition.

3.2 Services

The second core CANS component are *services*. Unlike drivers that represent rigidly constrained, mobile, soft-state adaptation functionality, services can export data using any standard internet protocol (e.g., TCP or HTTP), encapsulate more heavyweight functions, process concurrent requests, and maintain persistent state. The different interface requirements of drivers and services stem from the observation that most current services distributed in the internet are legacy in nature: their source code is general unavailable, and rewriting or modifying them is impractical. The price paid for not adhering to a standard interface is that unlike driver migration, CANS does not explicitly support service migration; a service individually determines how it manages its own state transfer. This design choice reflects the view that services are migrated infrequently and doing so requires protocols that are difficult to abstract cleanly.

CANS provides applications with a general platform to create, compose, and control services across the network. A service is required to register itself identifying the data types it supports, optionally providing a *delegate object* that can control the service and act on its behalf in interactions with the rest of CANS. The delegate object implements a standard interface consisting of activating and suspending the service, and receiving CANS events. Service composition is similar to driver composition, using types supplied at registration time.

3.3 Communication Adapters

Communication adapters are auxillary CANS components, which transmit data *physically* across the network to connect drivers that span different nodes. To achieve this, these components expose the same `DPort` interface, appearing to other drivers just as a regular driver. Communication adapters also support two additional kinds of logical connections: (1) between applications and drivers; and (2) between a driver and a service that exports data using an interface other than `DPort`.

To provide the above functionality, adapters establish physical communication links between application wrappers (see below) and execution environments, between two execution environments, and between an execution environment and a service. Multiple logical connections can be multiplexed on this single physical link; the latter can exploit transport mechanisms best matched to the characteristics of the underlying network. Communication adapters can additionally encapsulate behaviors that permit them to adapt to and recover from minor variations in network characteristics. For instance, these adapters can be written to use one of several network alternatives, automatically transitioning between them to improve performance. The continuity semantics upon such reconnection are dictated by the requirements of the data types associated with the adapter's ports.

3.4 Support for Legacy Applications

The CANS infrastructure supports both CANS-aware and CANS-oblivious applications. The former just hook into the driver and service interfaces described earlier. The latter require more support but are easily integrable because of our focus on stream-based transformations on the data path. Our solution relies on an *interception layer* that is transparently inserted into the application and virtualizes its existing network bindings. The interception layer is injected using a technique known as API interception [11], which relies on a run-time rewrite of portions of the memory image of the application.

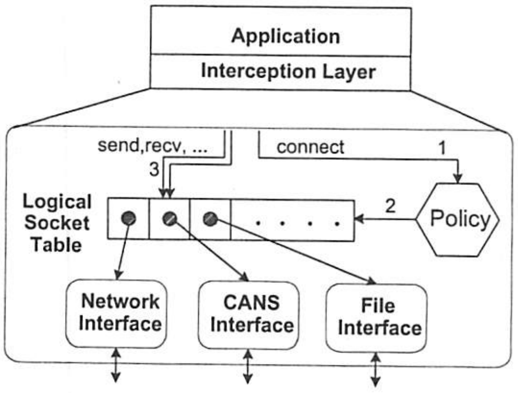


Figure 4. Architecture of the interception layer.

The general architecture of the interception layer is shown in Figure 4. The interception layer provides the application with an illusion of a TCP socket which can be bound to various interfaces (CANS or native network) for actual data transmission. An application specific policy responds to events (such as connect requests) delivered to it by the interception layer, which in turn influences the binding. Thus, enabling CANS support for a new legacy application would require only writing a specific policy for that application. Finally, although our current implementation virtualizes the TCP layer, the technique can as easily support other well-known protocols, such as HTTP.

4 Distributed Adaptation in CANS

CANS supports three modes of adaptation in response to dynamic changes in system characteristics: (1) *intra-component adaptation*, where each service or driver detects and adapts to minor resource variations on its own; (2) *data path reconfiguration and error recovery*, where the data path undergoes localized changes involving insertion, deletion, and reordering of drivers; and (3) *replanning*, where existing data paths are torn down and new ones constructed to respond to large-scale system variations. These three modes represent different points on the cost-functionality spectrum, enabling the system to respond to system events with the least overhead possible. To the best of our knowledge, CANS is unique in providing system support for data path reconfiguration.

4.1 Intra-Component Adaptation using Distributed Events

Each CANS driver and service can incorporate its own adaptation behavior that may or may not be coordinated with adaptation in other components. For example, a frame-dropping component can alter its policies upon detecting different levels of back-pressure on its output buffers. Note that adaptation in a single component is completely isolated as long as its effect is restricted to be within a single semantic segment (see Section 3.1).

To trigger adaptation, CANS provides distributed event propagation support, permitting components (including delegate objects for legacy services) to raise arbitrary events as well as listen for specific ones. Event support is realized by a per execution environment **Event Manager**, which is responsible for catching, firing, and transmitting events across the network. Event raising and firing is implemented using simple method calls and callback functions associated with the relevant component.

There are two major types of CANS events: events from the local resource monitor, indicating a change in resource status, and events from components on the data path. The first kind of events are sent only to local components that register themselves as interested listeners. The second kind, issued by components along a data path, are first sent to the *plan event delegate* (see Section 4.3), which is responsible for propagating the event along the data path as well as handling plan-specific events, such as events to trigger replanning.

4.2 Data Path Reconfiguration and Error Recovery using Semantic Segments

Insertion, deletion, or reordering of drivers along an active data path provides great flexibility in responding to a range of resource variations and link/node failure. However, a fundamental problem is that any such reconfiguration must preserve application semantics. In this paper, we focus on maintaining semantic continuity and exactly-once semantics. Specifically, any scheme must take into account the fact that the portion of the data path affected by the reconfiguration can have stream data that has been partially processed: in the internal state of drivers, in transit between execution environments, or

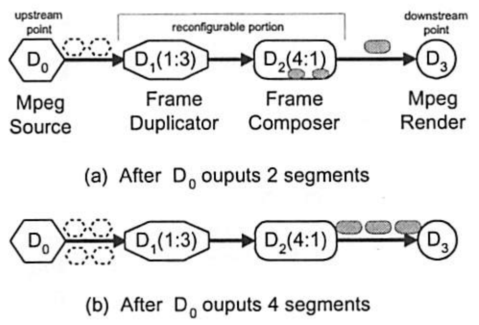


Figure 5. An example of data path reconfiguration using semantics segments.

data that has been lost due to failures. Note that although the soft-state requirement discussed in Section 3.1 permits us to restart a driver, it does not provide any guarantees on semantic loss or in-order reception.

Figure 5 shows an example highlighting this problem. To introduce some terminology, we refer to the portion of the data path that needs to be reconfigured because of a change in system conditions on the physical nodes or links (failures are an extreme example) as the *reconfigurable portion*, and the components immediately upstream and downstream of this portion with respect to the data path as the *upstream point* and *downstream point* respectively.[1] In the example, driver d_0 is a source of MPEG data, driver d_1 is an MPEG frame duplicator which produces 3 frames for each incoming frame, driver d_2 is an MPEG frame composer which generates one MPEG frame upon receiving four incoming frames from d_1, and d_3 is a renderer of MPEG data. The reconfigurable portion consists of drivers d_1 and d_2. Consider a situation where system conditions change after the upstream point d_0 has output two frames, and the downstream point d_3 has received one frame. At this point, the data path portion containing d_1 and d_2 cannot be reconfigured because doing so affects semantic continuity. The reason is that because of partially processed data in that portion, it is incorrect to retransmit either the second segment from d_0 whose effects have been partially observed at d_3, or the third segment, which would result in a loss of continuity at d_3.

The CANS infrastructure supports semantics preserving data path reconfiguration and error recovery by leveraging two restrictions placed on driver functionality, specifically semantic segments and soft state (see Section 3.1). Informally, the first restriction permits the infrastructure to infer which segments arriving at the

[1]For simplicity, we restrict our description to reconfigurable portions that have exactly one upstream and one downstream point. However, the solution is easily extendable to more general structures.

downstream point of the reconfigurable portion depend on a specific segment injected at the upstream point and vice-versa, while the second makes it always possible, even if any internal driver state is reset, to recreate the same output segment sequence at the downstream point by just retransmitting selected input segments at the upstream. Our solution exploits these characteristics to provide the required guarantees by just combining buffering and delayed forwarding of semantic segments at the upstream and downstream points respectively with selective retransmission of segments that are incompletely delivered. The correspondence between upstream and downstream segments is completely determined by driver characteristics in the reconfigurable portion; the implementation just needs to track marker messages that demarcate segment boundaries.

This scheme uniformly handles both the situation where drivers continue error-free operation but the data path needs to be reconfigured in response to system conditions, as well as the situation where link or node errors cause partial driver state to be lost. For the first situation, we defer reconfiguration to the time when the system can guarantee continuity and exactly once semantics. When some CANS events trigger reconfiguration, the upstream point starts buffering segments while continuing to transmit them, in effect flushing out the contents of intermediate drivers. The downstream point monitors the output segments arriving there, waiting until it *completely receives an output segment from upstream satisfying the property that all subsequent segments correspond only to input segments from upstream point either buffered at the upstream point or not yet transmitted.* At this time, the system can be stopped and the reconfigurable portion replaced by a semantically equivalent set of drivers. To restart, the upstream point retransmits starting from the first segment whose corresponding output segment was not delivered.

The same basic scheme also permits error recovery on portions of the data path that can be tagged a priori as possible sources of failure. The upstream point by default buffers all input segments before passing them on. The downstream point delays passing to the downstream driver any output segments that cannot be reconstructed in their entirity from input segments that are buffered at the upstream point, effectively isolating the downstream drivers from any duplicates that might get produced due to retransmission. When it is safe to pass on an output segment, the corresponding buffered input segments can be discarded. Upon an error, the affected components are re-instantiated, any buffered output segments at the downstream points discarded, and retransmission resumed from the first input segment whose corresponding output segment was never observed by the downstream driver. This scheme can be trivially extended to permit error recovery on portions that include services with checkpoint/restart facilities: the service needs to checkpoint whenever it produces a segment that corresponds to an input segment boundary.

In our example, reconfiguration works as follows:

1. The upstream point (d_0) starts buffering every segment it sends out after this time.
2. When downstream point (d_3) receives a complete segment from the upstream point (in this case this happens the third segment output by d_2 is received), it raises an event to the plan manager.
3. The plan manager can now freeze d_0, and replace d_1 and d_2 with a compatible driver graph.
4. To restart, d_0 retransmits starting from segment 5. In this case d_3 does not need to discard anything.

Error recovery on this portion requires d_0 to buffer its output segments and have the downstream point pass on segments to d_3 only in units of 3 segments at a time.

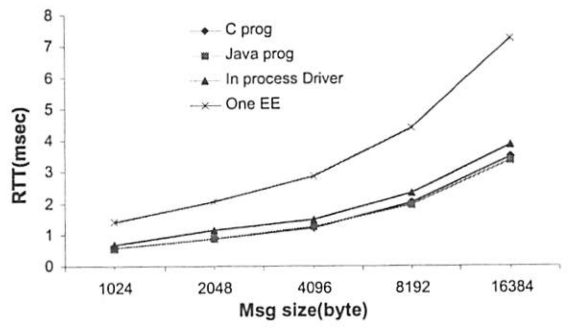
4.3 Planning and Global Reconfiguration

A plan refers to the deployment of drivers, services, and data paths in response to a request from a client application to connect to an end service. The key component responsible for planning in CANS is the *plan manager*, which is triggered when the interception layer detects a connect attempt on a TCP socket of interest. The plan manager takes responsibility both for creating the original plan, as well as changing it as required based on evolving system conditions. Such replanning is a last resort; as stated earlier, most changes are expected to be handled either entirely within a component or through localized data path reconfiguration.

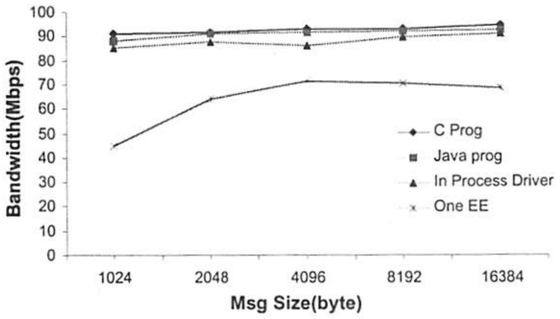
The planning procedure consists of two steps: *route selection* where a graph of nodes and links is selected for deploying the plan, and *driver selection* where appropriate drivers and services are mapped to the selected route. Space considerations prevent us from describing the steps in full detail, so we just highlight the overall strategy restricting our attention to plans that involve a single source and a single sink.

Route selection can be viewed as the shortest path problem in the node graph, which takes into consideration bandwidth on links between nodes in different domains and the relative loads on nodes within the same domain.

Driver selection bridges source and sink types while (1) efficiently using link and node capabilities along the selected route, and (2) overcoming problems caused by link properties such as insecure transmission and packet loss. The first subproblem amounts to selecting type-compatible components to construct the data path such



(a) Round Trip Time



(b) Bandwidth

Figure 6. Latency and bandwidth impact of the CANS infrastructure.

that node and link capacities are not exceeded, and some overall path metric (e.g., throughput) is optimized. The second subproblem imposes restrictions on the stream type at various points in the data path; for example, encrypted data is required in order to cross an insecure link if the sink requires a secure stream.

Our scheme unifies these two subproblems by defining the notion of an *augmented type*: each data type is extended with a set of link properties (e.g., security, reliability, and timeliness) that can take values from a fixed set. Network links are modeled in terms of the same properties and have the effect of modifying, in a type-specific fashion, values of the corresponding properties associated with different data types. To consider an example of HTML data transmitted over an insecure link, the data type represented by `HTML`(*secure*=`true`) is modified to `HTML`(*secure*=`false`) upon crossing a link with property *secure*=`false`. As a refinement to this base scheme, some data types have the capability to *isolate* others below them in the type stack associated with a stream from having their properties be affected by a link. For example, the `Encrypted` type isolates the *secure* property of types that it "wraps", i.e., encrypted data still remains secure after crossing insecure links.

Thus, the inputs to the driver selection process are the augmented type at the data source, the augmented type required at the sink, and the selected route (whose links may transform augmented types as described earlier). We use a dynamic programming algorithm to simultaneously select a component and map it to the route in a fashion that optimizes overall throughput. The partial solutions that make up the algorithm essentially look at the problem of converting the source type to an intermediate type on a subset of the route using only a fixed number of components. The complexity of this algorithm is $O(n^3 \times m^3)$, where n is the number of the components and m is the number of nodes.

5 Experience with Using CANS

We have been experimenting with a prototype CANS implementation on Windows 2000 clients and Java capable intermediate hosts, which currently emphasizes functionality and correctness over performance. Both the execution environment (EE) and driver components are written in Java. The interception layer described in Section 3.4 makes use of the Detours toolkit [11] to divert required application functions by rewriting portions of the memory code image. To set up the plan, the interception layer interacts with the plan manager on a distinguished EE, which in turn builds the plan, partitions it, and downloads plan fragments to individual environments. Interactions between different EEs make use of Java/RMI. Data transmissions between components, which are more performance critical, makes use of the communication adapters described in Section 3.3.

In this section, we first describe microbenchmarks reflecting overheads of using the CANS infrastructure, and then a larger case study that evaluates its flexibility.

5.1 Microbenchmarks

All measurements below were taken on a set of Pentium II 450Mhz, 128 MB nodes, running Windows 2000 and connected using 100 Mbps switched Ethernet.

Figure 6 shows the overheads introduced by CANS, measured in terms of how they impact communication between an application and an end service. Each graph shows the round-trip time and bandwidth achievable for different message sizes for four configurations: C prog and Java prog refer to our baselines, corresponding to application and server programs that communicate directly using native sockets in C or Java respectively. In process Driver and One EE refer to basic CANS configurations; the former shows the case when null drivers and a communication adaptor are embedded into the application interception layer and indicates the basic over-

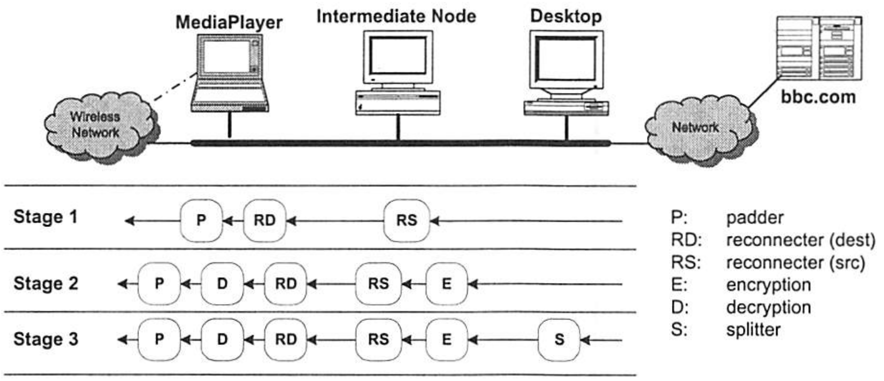


Figure 7. Case study: MediaPlayer with CANS infrastructure and the components added in each stage.

heads of driver composition, and the latter considers the case where the data path includes null drivers on an intermediate host between the application and service.

Figure 6 shows that the In process Driver configuration introduces minimal additional overheads when compared with the Java prog configuration (less than 10% arising from extra synchronization and data copying), attesting to the efficiency of our driver design and composition mechanism. On the other hand, the One EE configuration does show marked degradation in performance, primarily because of context switch costs and the fact that the transmitted data has to traverse across application-level and network-level four times instead of two times. However, given that intermediate EEs are intended to be used across different network domains where other factors dominate latency and bandwidth, this overhead is unlikely to have much overall impact.

5.2 Case Study

To evaluate whether CANS provides enough flexibility to support large-scale applications, we conducted a case study involving a shrink-wrapped application: Microsoft MediaPlayer. Our objective was to see whether CANS could be used to improve user experience with the application in a dynamically changing network environment *without* requiring explicit user participation (see Figure 7). The case study highlights CANS capabilities for (1) automatically selecting and deploying components suited to different network characteristics, driven solely by high-level type specifications at the source and sink, (2) dynamic and distributed event-driven adaptation upon detecting a change in system conditions, and (3) integrating with legacy applications and services.

The experimental environment (see Figure 7) consists of the client application run on a laptop with both wireless and wired network interfaces, a desktop capable of hosting services, an intermediate computer capable of hosting an execution environment, and an internet-based server providing media content (in our case, this was the bbc.com server). The case study consists of three stages: the laptop starts off being connected to the network using its wired interface, then is disconnected from the wired LAN, and finally physically moved away from the wireless access point. The transition from wired to wireless LAN is accompanied by a loss in security properties as well as a drop in bandwidth, which becomes worse in the third stage. CANS seamlessly insulates the client application from all network changes, continuing to seamlessly provide the user with the best experience afforded by underlying network characteristics.

CANS achieves this behavior by dynamically deploying appropriate components from a predefined set according to the planning algorithm described in Section 4.3. Note that route selection in this case is trivial, the one route involving all of the machines shown in Figure 7. The planning algorithm takes as input four pieces of information: the type definitions, the set of components, links modeled in terms of their link properties, and rules governing how types are affected by links:

- Figure 8 shows the data type definitions. `BaseStream` is the basic stream type with three boolean link properties, *reliable*, *secure* and *realtime*. `RStream`, `Media`, and `Encrypted` extend the `BaseStream` type, representing reliable, media, and encrypted streams respectively. `Video` and `Audio` are two subtypes of the `Media` type.
- Figure 9 lists the input/output types of six components, whose input/output types are listed in , along with the types produced by the source, *bbc.com*, and that required by the sink, MediaPlayer. Of the six, the *splitter* and the *padder* are Windows Media SDK based services; the first splits a video+audio ASF stream into an audio-only ASF stream available via HTTP, while the second "fills in" legal media frames whenever its input stream stops. The other four components are drivers, which cooperate

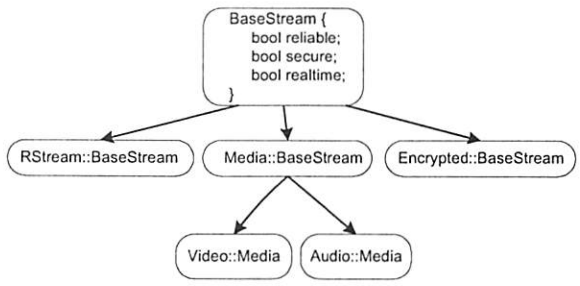


Figure 8. Hierarchical type definition of case study.

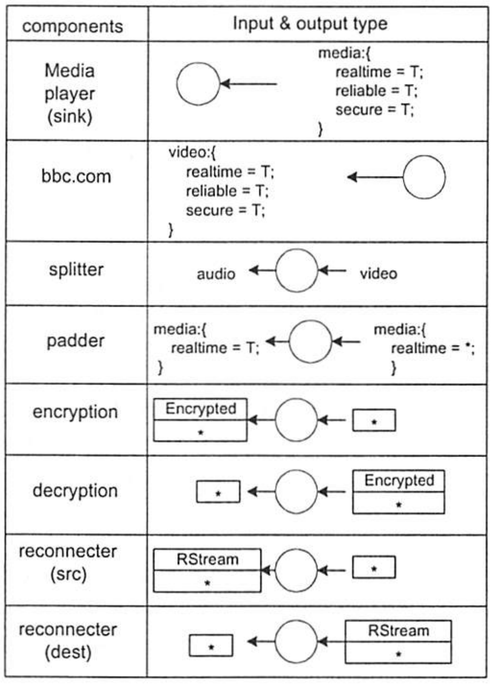


Figure 9. Input and output of case study components.

to handle encryption (*encryption* and *decryption*) and reliable transmission (*reconnecter(src)* and *reconnecter(dest)*) respectively.

- Table 1 shows the properties of the wired and wireless links and Table 2 shows how these effect different types. In Table 2, "effect isolation" refers to a type isolating the effect of a link property for data type instances below it in the type stack.

Figure 7 also shows the deployed components for each of the stages. For Stage 1, whose deployment is triggered when the application issues a connect call to an external streaming service, the plan consists of the *padder* and *reconnecter(dest)* running on the laptop, and *reconnecter(src)* running on the intermediate EE. These components are selected so as to guarantee the real-time and reliability requirements of the type expected at the sink. Not shown are communication adapters required for hooking up the application with the EE, and the EE

	properties		
	secure	reliable	realtime
wired	T	F	F
wireless	F	T	F

Table 1. The properties of links used in case study.

	secure		reliable		realtime	
	T	F	T	F	T	F
Media	—	F	—	F	—	F
RStream	—	F	T*	T*	—	F
Encrypted	T*	T*	—	F	—	F

—: no change *: effect isolation

Table 2. The effects of link properties on data types.

with the server, *bbc.com*. Needless to say, MediaPlayer receives a continuous data stream via the CANS components and is able to render it without any problems.

Stage 2 starts when we disconnect the laptop from the wired LAN. The *padder* ensures that MediaPlayer continues to receive legal frames even when no data is coming from the network. The communication adapters running on the laptop and the intermediate host detect the disconnection and reconnect to each other using the wireless interface, with data continuity at the semantic segment level ensured because of the *reconnecter(src)* and *reconnecter(dest)* drivers. At this time, because the wireless link has the *secure* property set to false, *encryption* and *decryption* drivers are installed into the data path automatically. Note that this adaptation involves the data path reconfiguration algorithm described in Section 4.2 to flush any in-transit segments.

Stage 3 starts when as the laptop is moved away from its access point, bandwidth drops below a threshold. The event detecting this triggers deployment of a new plan, resulting in the instantiation of the *splitter* component, capable of reducing stream bandwidth requirements. However, *splitter* supplies data of type `Audio`, which while compatible with the type specifications, `Media`, of the client application and other deployed components, requires the application to be placed in a different mode. Achieving the latter requires us to go outside the CANS infrastructure. Currently, we set up an ASX file that forces MediaPlayer to reconnect when the first connection is shutdown. This reconnect request is trapped and used to initiate the new plan. While this works, it also points out the need for a better abstraction of the protocol between applications and the infrastructure. Recent work by Lara et al. [4] of application adaptation relying on component automation interfaces points to what might be a promising direction.

Overall, the case study successfully demonstrated all of the important features supported by the CANS infrastructure: dynamic type-based composition and planning, event-driven adaptation that spans multiple drivers, and integration of legacy applications and services.

6 Related Work and Discussion

CANS shares its goals with many recent efforts that have looked at injecting adaptation functionality into the network. Instead of describing each separately, we group related efforts to put our work in perspective.

Adaptation functionality can be introduced only at the end-points or could be distributed on intermediate nodes. Odyssey [15], Rover [12] and InfoPyramid [14] are examples of systems that support end point adaptation. Each system provides only minimal support for composing adaptation activities across multiple nodes, and consequently may not be flexible enough to cope with changes in intermediate links. Efforts targeting adaptation at intermediate nodes in the network can themselves be viewed in terms of two issues: whether adaptation functionality is application-transparent or application-aware, and whether the functionality is introduced at the network level or the application level.

Systems such as transformer tunnels [16], protocol boosters [13] are examples of application-transparent adaptation efforts that work at the network level. Such systems can cope with localized changes in network conditions but cannot adapt to behaviors that differ widely from the norm. Moreover, their transparency hinders composability of multiple adaptations. More general are programmable network infrastructures, such as COMET [3], which supports flow-based adaptation, and Active Networks [17, 19], which permit special code to be executed for each packet at each visited network element. While these approaches provide an extremely general adaptation mechanism, significant change to existing infrastructure is required for their deployment.

Similar functionality can also be supported at the application level. The cluster-based proxies in BARWAN/Daedalus [6], TACC [7], and MultiSpace [9] are examples of systems where application-transparent adaptation happens in intermediate nodes (typically a small number) in the network. Active Services [1] extends these systems to a distributed setting by permitting a client application to explicitly start one or more services on its behalf that can transform the data it receives from an end service. A different perspective is offered by systems such as Conductor [20], which automatically deploy multiple application-transparent adaptors along the data path between applications and end services. Although such systems retain backward compatibility with existing applications, the lack of application input limits their flexibility. Furthermore, such systems rely upon self-describing properties of data streams, a condition that may or may not hold given increasingly proprietary content. More general are systems such as Ninja [8], PIMA [2], and Portolano [5], which permit construction of programmable ubiquitous access systems from networked services and transformational components. CANS also provides application-level support for injecting application-aware functionality into the network, but differs from the above systems in its focus on infrastructural support required for dynamic adaptation.

CANS has been most heavily influenced by the Conductor design and shares several features with the Ninja infrastructure. Conductor [20] provides an application-transparent adaptation framework that permits the introduction of arbitrary adaptors in the data path between applications and end services. Applications are integrated into the framework by modifying the kernel to trap calls that create and use TCP sockets. CANS borrows the idea of transparent stream-based adaptation from Conductor but differs in applying it to application-aware adaptation in a larger context that involves multiple services contributing to the data path; consequently, we require infrastructural support for downloading component code, instantiating the components, and ensuring compatibility. Also different is the degree of support provided by the infrastructure for reconfiguring existing paths, specifically the notion of semantics-preserving adaptation that spans multiple drivers, and general support for dynamic run-time composition of components.

Ninja [8] is a general architecture for building robust internet-scale systems and services consisting of three components: services, units, and paths. We restrict our attention to how paths are constructed in Ninja since that is the closest to our objective. Several CANS concepts find close matches in the Ninja design: our service-driver distinction is closely related to Ninja's service-operator distinction and both systems share ideas such as type-based composition and dynamic service adaptation. Despite these high-level similarities, the systems differ significantly in the details. Unlike Ninja, the CANS infrastructure provides support for (1) efficient composition of multiple drivers within the same physical host, (2) planning algorithms that consider route characteristics in addition to bridging type incompatibilities, (3) dynamic and distributed event-driven adaptation on existing paths, and (4) support for semantics-preserving adaptations that span multiple drivers; Ninja requires applications to provide their own mechanisms to ensure semantics such as guaranteed or in-order data delivery. On the flip side, it must be noted that unlike Ninja, CANS currently provides little support for scalability.

7 Conclusions

This paper has presented an application-level infrastructure, CANS, for injecting application-specific functionality into the data path connecting applications and end services. Such functionality can monitor and adapt to resource changes, providing the basic support needed for building novel application-level services that can seamlessly integrate diverse end devices across heterogeneous networks. Main contributions of CANS include: (a) efficient dynamic composition of components by requiring they adhere to a restricted interface, (b) dynamic and distributed adaptation using distributed events and novel path reconfiguration algorithms, and (c) support for legacy applications and services. Our experience indicates that CANS conveniently permits dynamic deployment and distributed adaptation of application-aware components to improve user experience.

CANS is one component of a larger project, Computing Communities, which focuses on distribution middleware for legacy applications. Our future work involves integrating CANS with related efforts emphasizing resource management and security issues, improving its performance, and designing better planning algorithms.

Acknowledgments

We thank the anonymous USITS reviewers for helping us improve this paper. This research was sponsored by DARPA agreements F30602-99-1-0157 and N66001-00-1-8920; by NSF grants CAREER:CCR-9876128 and CCR-9988176; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

References

[1] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proc. of the SIGCOMM'98*, August 1998.

[2] G. Banavar and et al. Challenges:an application model for pervasive computing. In *Proc. of the Sixth ACM/IEEE Intl. Conf. on Mobile Networking and Computing*, August 2000.

[3] A. T. Campbell and et al. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, April 1999.

[4] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. Technical Report TR-00-360, Computer Science Department, Rice University, October 2000.

[5] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges:data-centric networking for invisible computing. the portolano project at the university of washington. In *Proc. of the Fifth ACM/IEEE Intl. Conf. on Mobile Networking and Computing*, August 1999.

[6] A. Fox, S. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using infrastructural proxies:lessons and prespectives. *IEEE Personal Communication*, August 1998.

[7] A. Fox, S. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, October 1997.

[8] S. D. Gribble and et al. The ninja architecture for robust internet-scale systems and services. *Special Issue of IEEE Computer Networks on Pervasive Computing*, 2000.

[9] S. D. Gribble, M. Welsh, E.A.Brewer, and D. Culler. The multispace:an evolutionary platform for infrastructual services. In *Proc. of the 1999 Usenix Annual Technical Conf.*, June 1999.

[10] J. Haartsen. Bluetooth– the universal radio interface for ad hoc, wireless connectivitity. *Ericsson Review*, 1998.

[11] G. Hunt. Detours: Binary interception of win32 funcdtions. In *Proc. of the 3rd USENIX Windows NT Symp.*, Settle, WA, July 1999.

[12] A. D. Joseph, J. A. Tauber, and M. F. Kasshoek. Mobile computing with the rover toolkit. *IEEE Transaction on Computers:Special Issue on Mobile Computing*, 46(3), March 1997.

[13] A. Mallet, J. Chung, and J. Smith. Operating system support for protocol boosters. In *Proc. of HIPPARCH Workshop*, June 1997.

[14] R. Mohan, J. R. Simth, and C.S. Li. Adapting multimedia internet content for universal access. *IEEE Transactions on Multimedia*, 1(1):104–114, March 1999.

[15] Brian D. Noble. *Mobile Data Access*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[16] P. Sudame and B. Badrinath. Transformer tunnels: A framework for providing route-specific adaptations. In *Proc. of the USENIX Technical Conf.*, June 1998.

[17] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *Computer Communications Review*, April 1996.

[18] U. Varshney and R. Vetter. Emerging mobile and wireless networks. *Communications of the ACM*, pages 73–81, June 2000.

[19] D. J. Wethrall, J. V. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proc. of 2nd IEEE OPENARCH*, 1998.

[20] M. Yavis, A. Wang, A. Rudenko, P. Reiher, and G. J. Popek. Conductor:distributed adaptation for complex networks. In *Proc. of the seventh workshop on Hot Topics in Operating Systems*, March 1999.

Dynamic Host Configuration for Managing Mobility between Public and Private Networks

Allen Miu
MIT Laboratory for Computer Science
aklmiu@lcs.mit.edu

Paramvir Bahl
Microsoft Research
bahl@microsoft.com

Abstract

The usage and service options of a pubic network generally differ from a private (enterprise or home) network and consequently, the two networks are often configured differently. The existence of both types of networks motivates our need to improve support and management of nomadic users who frequently roam between them. We describe a solution that allows client devices to configure themselves dynamically to adapt to the local network configuration. In addition to supporting mobility, we describe how our solution also provides fail-over mechanisms for providing highly available service, load balancing, and location services. Furthermore, our solution can be used to scale networks that are deployed in a large setting. We discuss in detail the various issues that need to be dealt with for achieving true device-level mobility, pointing out several unsolved problems in this area. The algorithms and software proposed in this paper have been implemented, are deployed, and are currently being used in a real-world public network that is operational at the Crossroads Mall in Bellevue, Washington.

1 Introduction

Today, our economy and businesses rely heavily on people having Internet connectivity. This combined with the observation that we have become a highly mobile society in which many of us invariably find ourselves spending a considerable amount of time in public places and at public events compels us to move in the direction of providing high-speed Internet connectivity everywhere we can.

We have built and deployed a *public* wireless network, called the *CHOICE* network (URL: http://www.mschoice.com), which provides individuals Internet access in public places such as shopping malls, airports, libraries, train-stations etc. Our network is based on the widely available IEEE 802.11b standards-based wireless LAN technology [2], which enables us to provide Internet access to authenticated users at speeds up to *25 times* greater than speeds offered by 2.5G and 3G cellular phone networks [1]. Additionally, our network offers policy-based services such as different levels of privacy and security, different amounts of bandwidth, and different location services all on a per-user basis. For the host organization our network provides protection against malicious users and options for detailed accounting and flexible charging. Our design is conducive to developing interesting location services such as location-based buddy lists, electronic in-building navigation, and timely shopping promotions.

The underlying protocol that enables many of the aforementioned features of the CHOICE network is the **P***rotocol for* **A***uthorization and* **N***egotiation of* **S***ervices*, or PANS. PANS is a novel lightweight protocol that facilitates (a) global authentication of users; users can be authenticated from anywhere in the world, (b) authorization, monitoring and management of network access for authenticated users, (c) enforcement of policies on a per-user basis, and (d) device auto-configuration for supporting users who roam between differently configured networks.

Typical usage scenarios for private and public networks are different, and consequently, these networks are generally configured differently. Large corporations tend to be extremely security cautious, taking an enterprise-centric approach where every user is governed by a single policy. User authentication is intended to prevent unknown persons from accessing internal private networks. Public networks are security cautious only to the extent the individual using the network is. The host organization's focus is on establishing the identity of a *previously unknown* user and then giving her access to the network, its resources, and other location services generally for a fee. Hence, tracking who is using the network, what services are being used and how much bandwidth is being used are important. Another difference is, while corporations generally have a high level of confidence and trust in their users (employees), public network operators have to guard against the network users who they might not know well. They need tools to protect themselves from malicious users who are only interested in bringing the network down.

In thinking through the different usage scenarios and studying several privately deployed networks that we know of, we concluded that corporations generally use some sort of a pre-configured shared key mechanism with hardware encryption to secure network access. Public networks on the other hand perform packet-level processing for both user-level authentication and privacy, and for offering different kinds of services, and keeping track of network use on a per-user basis. Consequently, **client devices have to change behavior according to the network they are accessing**. When accessing the private network (normal mode), the client need not do anything; hardware encryption with a shared key is sufficient to control users' access. However, when accessing the public network (special mode), the client runs through an authentication process and starts using a specialized network access protocol (e.g. PANS), which gets it different types of interesting services.

With these issues in mind, we developed a mobility support mechanism that allows devices to automatically determine how to establish/re-establish network connectivity as roaming users migrate across the different networks.

We present the architecture and operation of the CHOICE network, focusing on the problem of supporting mobility at the device configuration-level. We briefly describe PANS and the features it provides, leaving out details that are documented in [3]. We show how our system's mobility architecture allows us to support other important features like load balancing, scaling, and location services.

The primary contribution of our work is a detailed design of a system and protocol that offers the following important features:

a. Dynamic configuration of client devices, without user intervention, as nomadic users roam between public and private networks.
b. Dynamic configuration extensions that support a fail-over mechanism as well as a scalable key-distribution system, and
c. Support for location services currently not available in other networks.

The rest of this paper is organized as follows: Section 2 sets the stage by describing a typical usage scenario of public and private networks. Section 3 describes the system components of the CHOICE network. Section 4 then articulates the precise mobility problem and in Section 5 and 6, we discuss our design criteria and our solution. In Section 7, we explain how our solution can be extended to help public networks achieve high availability and scalability. We discuss on-going and future work for achieving true mobility in Section 8 and we survey related work in the field in Section 9. Finally, we conclude in Section 10.

2 A Typical Usage Scenario

A person walks into a public place where she has arranged to conduct a business meeting with another person from a different company. Both people are savvy wireless LAN users and come equipped with their notebook computers and wireless LAN cards. The public place has a CHOICE network that is available to the general public for a small fee. As the user sits down at a coffee table waiting for her companion, she switches-on her notebook computer, launches her web browser, and points it to http://choice. If she has not already done so, the user downloads the network access software (PANS client) from the local web server and installs it on her notebook. A reboot of the machine is not required for this installation. Upon installation, the PANS client module detects the presence of the CHOICE network and displays a welcome message to the user indicating to her that she can get Internet access by logging on and establishing her identity. The user then proceeds to authenticate herself via the local organization's log-on page wherein she types in her identity and password. These are sent to a global authentication database to which the local host organization subscribes. When the user's identity is established and authentication granted, the network checks to see the policy that is to be applied for this particular user (e.g. how much bandwidth to give, what security level to grant and how much to charge, default values exists for first time users). Based on policy, the network generates a unique key and sends it to the PANS client. At this point the user's web browser automatically refreshes to the local portal and Internet access is now possible.

After she is done with her meeting she log-offs and the network provides her with some usage statistics. She returns to her company and opens up her notebook, which she had placed in "hibernate" mode. As the notebook turns on, the PANS client senses that a different network is present and stops all special processing that is necessary for the CHOICE network.

We now describe the components of CHOICE and then explain the mobility problem more precisely in the following section.

3 Overview of the Choice Network

The CHOICE network has several system components that manage address allocation, authentication, authorization, security, accounting, and last-hop QoS. Figure 1 illustrates the different components of the CHOICE network as it has been deployed at the Crossroads Mall, Bellevue, Washington. Our description of the CHOICE network will be brief as we refer the reader to [3] for a detailed description of PANS.

3.1.1 Address Management and Naming

The CHOICE network uses a standard DHCP server to lease IP addresses to potential clients. The IP address scope and the lease period are configured by the host organization at setup time. Where DHCP's limited scope is an issue, a Network Address Translator (*NAT*) [5] is used instead.

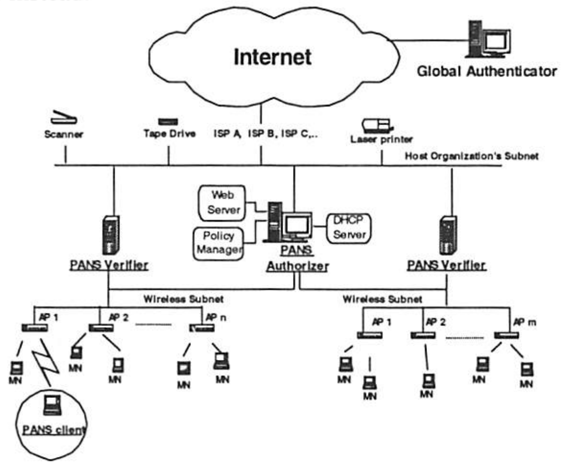


Figure 1: The various components of the CHOICE network as deployed at the Crossroad Mall in Bellevue, Washington

The IP address is given out even before the user has been authenticated to allow her access to local portals and to allow her an opportunity to download the network access software if she hasn't already done so.

The web server is the user's entry point into the CHOICE network. The local network web server is based on Active Server Pages (*ASP*) [6] and guides the user through the authentication process.

3.1.2 Authentication Database

Ideally, the CHOICE network authenticates a user by requesting from her a signed certificate containing her credentials. Thus, the CHOICE network can directly confirm the user's identity, assign the appropriate service policies for the session, and connect the user to the network seamlessly without asking for a password.

Unfortunately, there are a couple of problems that need be resolved before the system can use personal certificates for user authentication. First, it requires every user to register with a certificate authority. Since our goal is to deploy and offer public wireless network services today, we have to consider an alternative approach that is generic and readily accessible. Second, it is difficult to revoke a certificate and modify user credentials in a timely fashion. In this case, the CHOICE network may not assign the correct service policies based on the latest set of credentials.

Our current solution to this problem is to use a global authentication service that is accessible throughout the Internet. Our system uses MS Passport [8] as the authentication database. Several factors motivated our choice of MS Passport. First, its wide availability enables us to offer network service to a substantial number of users. Second, all transactions with Passport are web-based thereby greatly enhancing the usability of the system for the layperson. Third, all transactions with Pass-

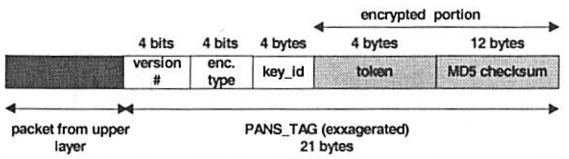


Figure 2: A key-tagged Packet. The version number, encryption type and `key_id` form the unencrypted portion, while the token and MD5 checksum are encrypted using the encryption algorithm specified under the encryption type.

port are carried out over *SSL* [9]. Thus there is an end-to-end secure channel between the user and the authentication service. Even if CHOICE were to be set up by an un-trusted third party, this party is not able to decrypt the user's name and password while it is being supplied to Passport.

3.1.3 The PANS Authorizer

The *PANS Authorizer* is a gateway to the global authentication server. To prevent unauthorized access to the Internet, the Authorizer performs IP-level filtering based on the destination IP-address of each packet. Any packet with a destination address other than the DHCP server, the WINS server, the DNS server, the local web server or the Passport server is dropped. Additionally, the Authorizer authorizes clients' access to the network upon successful completion of user authentication. It handles the task of determining service policies, generating per-user session keys, and distributing the keys to the clients and to the service gateways (i.e. *PANS Verifiers,* to be discussed next).

Each session key is valid for a finite amount of time. Depending on the host organization's preference, the key can either be automatically renewed or the user can be forced to explicitly obtain a new key when the present one is about to expire.

Once the user has been authenticated, all her communication is directed to the assigned service gateway. Individual packets are key-tagged (see Figure 2) by the client and verified by the service gateways to ensure that only authorized traffic is allowed to gain access to the Internet.

3.1.4 The PANS Verifier

The PANS Verifier handles the tasks related to per-packet verification, accounting and policy enforcement on packet transmissions between the mobile users and the public network. The PANS Client uses the Verifier as a service gateway for Internet access. The Verifier checks each packet for a valid tag generated by the client's session key. In addition, the Verifier keeps an account of the number of packets per user it has serviced and enforces policies such as QoS service-level by dropping packets from a user who violates her service agreement.

Because the task of the Authorizer and the Verifier are separated, multiple Verifiers may be deployed to handle large volumes of traffic flow within a subnet of wireless access points. Additionally, Verifiers may be replicated to support roaming between different subnets.

3.1.5 The PANS Client

The PANS Client resides on the mobile user's device. Once the Authorizer has granted access and downloaded the key to the client machine, the PANS Client tags every outgoing packet before sending it to the Verifier (see Figure 2). Depending on the level of service the user has opted for (which may be pre-configured into the machine or arranged dynamically), the PANS client can optionally encrypt the entire packet or only a portion of the outgoing packet. The Verifier can then decrypt the packet, and remove the tag before forwarding it on to the network.

The PANS client tags packets only when the public network service is present. The client host may use the same key when it migrates to a different subnet but must negotiate a new key when it migrates to a different public network.

3.1.6 Performance

The task of per-packet verification by decrypting a packet and checking for a valid signature puts a limit on the number of connections the PANS Verifier can handle. To determine this limit we ran several tests that measured the network throughput, CPU utilization, and packet round trip time (RTT) with PANS enabled. A detailed description of the experimental methodology and analysis of the results is provided in [3].

We flooded a Verifier with PANS packets via a 100 Mbps Ethernet link and found that for bulk transfers, the network saturates before the CPU does. With the link completely saturated we observed that on average the network throughput decreased by 10% and the CPU utilization increased by 40% in the presence of PANS processing. On flooding the network with 100,000 64KB-buffers and varying the packet size during each run, we found that the per-packet RTT difference between connections with PANS enabled and without PANS was in the order of tens of microseconds.

Overall a single Verifier, which in our experiments was a 450 MHz Pentium II Dell Precision 410 workstation with 128 MB RAM, can easily handle traffic from ten 11Mbps wireless access points (APs) with PANS enabled.

4 The Mobility Problem

While PANS provides a protocol to authenticate clients and a means to control user access privileges, it does not specify any mechanism for discovering the PANS service, or a scheme for managing the client's configuration according to the available access mode in the network. To illustrate where these problems arise, let us examine the following three sample scenarios:

1. The client host migrates between the company private network and the public network. Since the company network may not be running PANS, the client host must recognize when to enable / disable the public network protocol locally.
2. The client host migrates between different subnets of the same public network. In this case, it would be undesirable to require the user to re-authenticate herself by repeating the logon process. Instead, the client should gain access in the new subnet by using the same key obtained from the previous subnet. The client host must recognize and perform any necessary changes in the routing configuration (e.g. directing traffic to a different Verifier server) and resume network operation by using the same key.
3. The client host migrates between different public networks. The client host must distinguish this from the previous scenario and ask the user to perform the logon process in the new network. After authentication has succeeded, the client host will use a new key to communicate in the new network. However, the host should save the previous key until it expires so that it could be reused upon returning to the previous network.

All three scenarios involve a combination of changing the client host's routing table, enabling / disabling the PANS module, and managing a set of keys acquired by the client. While one can change these configurations manually when the client host is relatively immobile, it would be painful, if not impractical, to have the user reconfigure the host every time she moves to a different network.

Before we describe how we solve these problems, we outline our design goals in the following section.

5 Design Criteria

The goals that influenced the design of the auto-configuration module for the CHOICE network are summarized below:

(I) Efficiency: Since our system will most likely run on wireless, mobile devices, the system should be lightweight and efficient in terms of bandwidth, memory, processing, and power.

(II) Responsiveness: The mobile host should detect a change of environment and self-configure within seconds.

(III) Ease of Deployment: We wish to avoid any changes in the existing protocol to support our auto-configuration system. Furthermore, we wish to avoid relying on any other special protocols to handle service discovery and auto-configuration. Our system should work with any

standard network stack commonly found in all types of mobile devices and operating systems.

(IV) Hardware agnostic: Our system should not require any modification to existing hardware. Also, the system should work in both wired and wireless networks.

(V) Privacy and Security: The auto-configuration system should not compromise the security and privacy models in the original PANS protocol. Namely, the auto-configuration system should ensure that the system is configured with safe and legitimate parameters.

(VI) Flexibility: We wish to examine whether employing a particular scheme would allow us to expand and implement additional features on top of PANS.

6 PANS Auto-Configuration Module

In this section we describe the requirements, design criteria, architecture, algorithms and the implementation details behind the mobility support module we have built for the CHOICE network.

6.1 Required Functionalities

The auto-configuration module needs to perform four basic functions to manage mobility between public and private networks: service discovery, bootstrapping, protocol configuration, and key management. We discuss each of these in detail below.

6.1.1 Discovery of the Public Network Service

To correctly configure the mobile host for public or private network accesses, the mobile host must first discover if a public network service is offered in the local network. To discover such a service, either a solicitation or beaconing technique may be used. We have chosen the beaconing technique for the following reasons:

- Beaconing is unidirectional so it cuts transmission overhead by half for the client host when compared to a bi-directional polling-response scheme.
- Beaconing consumes only one unit of transmission time per broadcast period, which is significantly less than the $2n$ units of transmission time consumed by n different clients in a polling-response scheme. In a wireless medium, a beaconing scheme reduces the airtime overhead and the level of traffic contention in the system.
- Polling drains more power not only from the wireless host that is broadcasting the probing messages, but also power from third-party wireless devices that must expend energy for receiving these messages broadcasted within the vicinity.
- Polling may introduce unwanted solicitation messages in private networks as client hosts continually probe for the public network service. Alternatively, a client can limit the number of broadcast queries by sending probes only when there is a good hint that the client may have migrated to another network (e.g. the hardware detects a link state change or when the host detects excessive amount of packet loss [10]). Unfortunately, such hints given by other network layers are often implementation dependent and consequently, unreliable.

6.1.2 Bootstrapping

Once a public network service is discovered via broadcasted beacons, the auto-configuration module should ensure that the mobile host has a valid IP address. (The ability to receive beacon broadcasts does *not* imply that the client has a valid IP address.) Then, the auto-configuration module sets the default gateway to the advertised Authorizer IP address, points the client's default web browser to the advertised URL of the local portal containing the authentication script, and prompts the user to begin the web-based authentication process.

6.1.3 Protocol Configuration

The auto-configuration module controls when the local PANS protocol driver should start or stop tagging and possibly encrypting/decrypting packets at the mobile host. This depends on whether the mobile host is inside a public or a private network. When the mobile host is in a public network, the default gateway must be set to the public network's Authorizer or Verifier, depending whether the user has been authenticated and obtained a valid session key. When the mobile host roams from one subnet to another within the public network, the mobile host must detect the migration and set its default gateway to a Verifier in the new subnet. When the mobile host leaves the public network, the local PANS driver should stop tagging packets and the default gateway should be set to the default system values (e.g. DHCP).

6.1.4 Key Management

After the user has been authenticated in the public network, she is given a session key for accessing the Internet. The session key expires after a pre-defined period. Over time, a user can enter and exit different public networks and collect a set of session keys. Therefore, whenever the user enters a public network, the auto-configuration system should determine whether the user currently has a valid key. If so, the system should automatically bypass the authentication procedure and pass the correct key to the local PANS protocol driver to access the public network. When the key is about to expire, the auto-configuration system should initiate a procedure for renewing the session key.

6.2 Architecture and Implementation

Due to the considerations listed above, we have designed and built an auto-configuration system that uses beacons to discover the PANS service and obtain the necessary configuration parameters to bootstrap the authentication process. When the client migrates out of the public network service, it no longer receives any beacons. The client times out and resumes normal networking operation by disabling the special mode at the local PANS driver and resetting the default gateway value.

Our scheme is very similar to the broadcast advertising schemes found in Mobile IPv4 and Mobile IPv6 [11] except that it also supports a number of other extended features such as client-side key management, a system-wide fail-over mechanism, and location-sensitive messaging. The next section describes the components and the algorithm of the system.

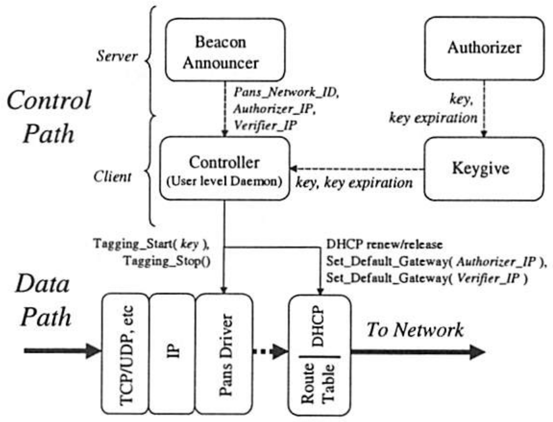

Control Path
Server
Client
Beacon Announcer
Authorizer
Pans_Network_ID, Authorizer_IP, Verifier_IP
key, key expiration
Controller (User level Daemon)
Keygive
key, key expiration
Tagging_Start(key), Tagging_Stop()
DHCP renew/release
Set_Default_Gateway(Authorizer_IP),
Set_Default_Gateway(Verifier_IP)
Data Path
TCP/UDP, etc
IP
Pans Driver
DHCP
Route Table
To Network

Figure 3 Architecture of the auto-configuration system for PANS

6.2.1 Components

Figure 3 illustrates the architecture for supporting auto-configuration in PANS. The diagram divides two types of data flow. The control path illustrates the flow of configuration parameters and control signals of the auto-configuration system. Sitting below the control path is the data path, which illustrates how data packets flow through the network stack incorporated with an intermediate PANS protocol driver for packet tagging.

Inside the control path are two types of modules. The server modules, which consist of the *Beacon Announcer* and the *Authorizer*, sit inside the Authorizer server and sends configuration parameters and session keys to the client modules. The Beacon Announcer periodically broadcasts a beacon, which contains a unique PANS network_id, a subnet_mask, a URL of the user log-on web page, the current Authorizer_ip and Verifier_ip addresses.

As mentioned in Section 4, the PANS Authorizer serves as a proxy for a global authentication server such as MS Passport. After the user completes the authentication process, the Authorizer establishes a secure *SSL* connection with the client's web browser. Using this connection, the Authorizer delivers the session key and key expiration values to the key manager inside the client's Controller daemon.

Although using the web browser's *SSL* service saves us from implementing a special secure protocol for key delivery, we now need a way to direct the session key from the web browser to the key manager. To do this, the ASP script on the Authorizer delivers the key values via a MIME-typed data stream, which triggers the web browser to launch the registered *Keygive* user level program[1]. The web browser then pipes the key values to the Keygive module, which in turn hands them over to the Controller.

Notice that the Verifier_ip values are deliberately transmitted inside the broadcast beacon instead of being transmitted alongside with the session key. This is done to allow those clients who have migrated to a different subnet but still hold a valid session key to directly access the local Verifier without repeating a full authentication process. Furthermore, such a design supports a useful system-wide fail-over feature and load-balancing mechanisms. For example, the *Beacon Announcer* can broadcast a different Verifier_ip to instantly migrate all the clients to use a backup gateway.

The heart of the auto-configuration system is the Controller, which runs a finite state machine to handle the external events generated by the server modules and to coordinate the tasks of discovering a public network service, bootstrapping the authentication process, configuring the PANS protocol driver and the routing table, and managing session keys on the client host.

The Controller listens to two well-known ports: one detects beacons coming from the Announcer, and the other receives the session key and expiration values from the Keygive module. The Controller stores the session key value into a table indexed by the network_id associated with each key. The Controller implements an earliest-expiry-time replacement policy and invalidates a session key entry whenever it expires. Then by matching a valid row entry with the currently advertised network_id, the Controller can use the appropriate session key to configure the local PANS driver via an *ioctl* call.

6.2.2 Operation

Figure 4 depicts the Controller's finite state machine. Bootstrapping starts when the Controller detects the first beacon. The Controller uses the network_id and the subnet_mask in the beacon to distinguish whether the client has roamed to a different subnet within the same network or migrated to a different public network. In either case, the Controller verifies that client has a valid IP address to operate in the new subnet and updates the

[1] By accepting keys via MIME-typed data streams, the Keygive program may be launched by malicious web portals. Although not currently implemented, we can easily extend the Keygive program to authenticate the key delivery channel via certificates or other secure mechanism such as S/MIME [25].

address if necessary. Our system currently relies on DHCP to obtain a dynamic address assignment and set the default DNS server. To ensure timely address assignment, the Controller will force a DHCP request and loops in the Detect state until the client host receives a valid address that is contained in the subnet advertised by the beacon. This is done to handle roaming problems where DHCP fails to recognize a network migration due to inconsistent media sensing implementations.

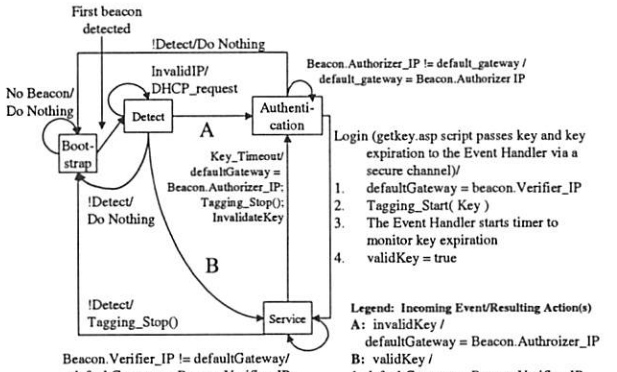

First beacon detected
!Detect/Do Nothing
InvalidIP/
DHCP_request
No Beacon/
Do Nothing
Boot-strap
Detect
Authenti-cation
A
Beacon.Authorizer_IP != default_gateway /
default_gateway = Beacon.Authorizer IP
Login (getkey.asp script passes key and key expiration to the Event Handler via a secure channel)/
1. defaultGateway = beacon.Verifier_IP
2. Tagging_Start(Key)
3. The Event Handler starts timer to monitor key expiration
4. validKey = true
Key_Timeout/
defaultGateway =
Beacon.Authorizer_IP;
Tagging_Stop();
InvalidateKey
!Detect/
Do Nothing
B
!Detect/
Tagging_Stop()
Service
Beacon.Verifier_IP != defaultGateway/
defaultGateway = Beacon.Verifier_IP
Legend: Incoming Event/Resulting Action(s)
A: invalidKey /
defaultGateway = Beacon.Authroizer_IP
B: validKey /
1. defaultGateway = Beacon.Verifier_IP;
2. Tagging_Start()

Figure 4 Controller Finite State Machine

Next, the Controller checks the key management table to determine if the client currently possesses a valid key for the current network by checking the beacon's network_id. If so, it bypasses the login and authentication procedures and sets the default gateway to the advertised Verifier. The Controller enters the Service state so that the client can seamlessly resume the previous PANS session. This setup supports client migration between subnets as well as migration between different public networks.

If the client does not have a valid key, the client has entered a new public service and must begin the authentication process to gain full access to the network. The Controller enters the Authentication state where it sets the client's default gateway to the advertised Authorizer IP address and extracts from the beacon the URL of the authentication web page. Then, the Controller pops up a greeting message to notify the user about the discovery of a new public network service and displays the URL. If the user wishes to join the service, the user launches her default web browser to the given URL, and begins the web-based authentications process. The Controller waits in the Authentication state until the authentication succeeds or until the client migrates out of the current network. When the authentication succeeds, the Controller enters the Service state by saving the session key into the key management table, setting the default gateway to the Verifier, passing the session key to the PANS driver, and enabling packet tagging via an *ioctl*.

The user gains full access to the Internet in the Service state. Service can be interrupted when the user migrates to another network, when the session key expires, or when no beacons are detected. In the case of user migration, we repeat the bootstrapping process. When the key of the ongoing session expires, the Controller negotiates for a new session key. The user may be prompted for a confirmation or the renewal process can happen automatically, depending on the user's preference. When no beacons are detected for a fixed period, the public network service is no longer available. The Controller disables the PANS driver and configures the client to the system's default parameters for accessing private networks.

As a final remark, we wish to emphasize how we have decoupled key management and mobility management so that the network access protocol (PANS) and the auto-configuration mechanism can work independently of each other. Our system includes all the network parameters within the beacon to support the bootstrapping process. Thus a client possessing a valid session key can immediately access the network without repeating the authentication process. Likewise, the Controller is free to refresh a client's key during an authenticated session without affecting the client host's network configuration. In Section 7, we will explain how this decoupling of key and mobility management also helps us implement a scalable key distribution scheme as well as fail-over and load balancing mechanisms for the public network infrastructure.

7 Beyond Mobility - Extending the System

One of the main goals of the CHOICE Network project is to deploy public network access service in large settings such as major conference centers, airports, shopping malls, and the like. Thus, the network access service must itself be scalable and highly available.

We have considered these issues when designing the auto-configuration mechanism for PANS, and have found ways to extend the beaconing mechanism to help the network access service attain scalability and fault-tolerance. We have found other applications for the beaconing mechanism as well. We will describe the various extensions we have considered in the next few subsections.

7.1 Auto-Configuration to Provide High Availability and Scalability

Because the Authorizer and Verifier contain the set of active keys in the network, the public network service must provide a fail-over mechanism to handle the case when a Verifier fails. To prevent loss of information, a service provider may install multiple backup Verifiers that replicate the table of keys currently active in the network.

In addition, service providers may scale their service by installing multiple Verifiers to share high traffic loads.

Hence, the requirement for achieving high availability and scalability is to configure each client to use the appropriate gateway when accessing the network.

We extend our auto-configuration system to support fail-over and load balancing as follows. For each beacon, we advertise a *vector* of Verifier_ip addresses instead of one Verifier_ip. The vector represents the set of operational Verifiers, which excludes the set of backup verifiers. The first element of the vector is the "preferred" gateway of which new clients should use when they successfully log on to the network for the first time. Thus, to balance the traffic load among the operational Verifiers, the advertised value for the first element is rotated according to the load of the network.

When a gateway fails, the Authorizer advertises the backup server as the last element of the vector and removes the IP address of the failed gateway from the vector. Thus, clients in the network continually scan the vector in every beacon to ensure that their gateway is available. If not, the client simply switches to the backup Verifier that is advertised as the last element in the vector. Because the backup Verifier contains a copy of all active keys in the Verifier that failed, the transition should occur smoothly, with minimal disruptions, if any, to all ongoing network transaction.

7.2 Subnetting and Scalable Key Distribution

For a number of administrative reasons, subnetting may be required to scale large public networks. Each subnet has its own address space and its own set of Authorizer and Verifier gateways. When a mobile client roams across subnets, the host must change its IP address and set its default gateway to the Verifier advertised in the new subnet. In Section 6, we have already discussed how our dynamic host auto-configuration system supports mobility for the client host. However, we have glossed over the issue about how the keys are to be distributed behind the network.

One simple solution is to distribute keys globally within the network infrastructure. However, this approach clearly does not scale well as the number of users grow in the network.

The requirement for scalable key distribution in the network infrastructure is to avoid sharing key information globally among all the Verifiers in the network. Each Verifier should be responsible for managing the set of keys belonging to the set of active clients in its own subnet. Under this requirement, the network must be able to migrate keys to the Verifier that handles traffic from the client's current location.

Here, the auto-configuration system can help. The Controller can keep a history of the subnet that the client has previously visited. Then when the client roams to another subnet, the Controller can automatically request a key migration from the Authorizer server in the new subnet. The request contains an encrypted portion containing the client's token and an unencrypted portion, which includes the client's key identifier and the address of the original Authorizer that issued the key. The Authorizer in the new subnet would forward the request to the original Authorizer, which authenticates the request by checking the encrypted token against the token stored in its table. If the verification succeeds, the original Authorizer will return the requested key to the new Authorizer via secure channels. After the new Authorizer receives the key, it redistributes the key among the Verifiers in the new subnet and acknowledges the client. Thus, the client migrates to a new subnet seamlessly by using this scalable, on-demand approach to key distribution.

7.3 Location Services

In a wireless network, the beaconing mechanism could also be extended to provide certain coarse-grain location information. We will outline two applications below.

7.3.1 Network Usage Service Metric

A very practical piece of information to include in the beacon is a metric that represents the network's load. For example, as the Verifier server becomes highly loaded, the Authorizer can advertise a low service quality metric to the clients. The Controller can be modified to interpret these metrics and notify the user about the current conditions of the network. Hence, users can change their expectations or access behavior according to the system's feedback. For example, if there is a cost associated with accessing the public network, then a user can decide whether it is worth the cost to register with a public network that is presently congested.

Finding an appropriate metric for this purpose is still an open problem. It is unlikely that the Authorizer or Verifier server would ever become the bottleneck of the system (provided that the administrator has scaled the system using the suggested techniques). Rather, the individual access points in the wireless network are more likely to become the bottlenecks. Hence, we can extract load information from each access point and redistribute this "metric" to the associated clients. To our knowledge, the availability of such load information varies between different access point implementations. It would convenient if the API for extracting such information were standardized.

7.3.2 Coarse level Location Information

Generally, device drivers of wireless network interfaces (WNIC) can obtain the id of the Access Point (AP) with which the WNIC is associated. If so, a client can download a map of all APs within the vicinity and use the id to locate its position on the map. A client can then infer that her location is roughly the same as the AP with which she is associated. Although this is a very coarse-grain approach for identifying the user's location when compared to the proposed alternatives [21][22][23], it is nevertheless a useful feature (especially in large settings such as the airport, where many APs are deployed) that can be readily implemented in our system.

Another simple but useful location-sensitive application is "coded messaging." Instead of mapping the access point identifier to a physical coordinate on a map, the Controller can map the identifier to a table of messages. Thus, depending on the user's preference, the Controller can pop up messages to notify the user about a special event that is happening near the access point of which the client is associated (e.g. notification of a special promotion at a nearby coffee shop). The table can also include a time-index so that messages can pop up at specific times during the day.

Finally, we would like to extend a word of caution. The purpose of this section is to illustrate the power behind a beaconing system and to illustrate how it could be used to build simple but useful services. It is not well-suited for implementing heavy-duty service discovery applications mentioned in [19][20]. In particular, we must be careful not to overload the beacon with too much data as our design goal is to keep the auto-configuration system lightweight. The examples above show how to do this by means of mapping compact codes contained in the beacon with a downloadable table containing the full information required for the application.

8 Discussions

In this section, we will discuss some issues that need to be considered for providing secure and seamless mobility support in our auto-configuration system.

8.1 Mobile IP vs. Auto-Configuration

We wish to emphasize that the set of mobility problems addressed by our auto-configuration system is different from those addressed by Mobile IP and other similar IP-level migration protocols. Mobile IP is primarily concerned with locating the mobile host and re-routing packets to the host's current destination. In contrast, our protocol is concerned with configuring the host to migrate between public and private networks.

Nevertheless, both systems do share some similarities. When the mobile host migrates to a foreign network, the protocol employs a similar beaconing strategy to probe for a Foreign Agent and configure the local Mobile IP stack to the correct mode of operation. Despite this similarity, we chose not to extend Mobile IP to support the auto-configuration requirements in PANS. Our goal is to be protocol agnostic so we avoided tying our system to any specific protocols. Hence, any protocol, including Mobile IP, will continue to operate seamlessly on our system.

8.2 Low-Level Configuration

There is one situation that may prevent our auto-configuration system from migrating a client between public and private networks. The problem is caused by special configurations in the WNIC. As mentioned in the introductory sections, some private networks use the wired equivalency protocol (WEP) to secure the wireless link [2]. A user must manually enable the WEP key settings in the WNIC driver, otherwise none of the beacon packets (IP-level broadcast packets) would reach the client host.

We are aware of some on-going efforts that specifically address this issue. For example, future mobile clients will automatically cycle through several different pre-configured WEP keys in an attempt to associate with the wireless network. When all keys fail, they will try to associate with the network with the WEP key disabled. With the appropriate extensions and API for supporting mobility in this manner, our system should migrate clients seamlessly between all types of public and private networks.

8.3 High-Level Configuration

Problems with application settings may arise as the user migrates between networks. For example, a client's web browser may default to a proxy server in the corporate network. After migrating to the public network, the user might find excessive browser delays caused by timeouts as the browser tries to locate the default proxy. To prevent such problems, the applications should be made aware of the host's mobility [25].

We should mention that another solution to this problem is to employ Mobile IP. However, using such techniques may induce certain limitations [24]. For example, Mobile IP, in certain cases, may tunnel packets destined for the mobile agent from the Home Agent. If the home agent is situated in the corporate network, the client traffic will be governed by the policies imposed by the corporate proxy. In contrast, the user may gain full access to the Internet via other end-to-end mobility mechanisms [25] that allow applications to open direct connections and assume the access policies defined by the public network.

8.4 Beaconing, Polling and Issues about Media Sensing in Wireless Networks

Before auto-configuration can be triggered, the client host must implement a mechanism to detect when it has migrated to another subnet or to another network. We solve this problem by comparing the network_id values between beacons. This is similar to the mobility detection algorithms proposed by Mobile IP [11].

Another common solution to the mobility detection problem is to rely on a link-level (a media sensing) mechanism to trigger the auto-configuration mechanism when there is a change in the client's link state. This scheme works well for DHCP in wired networks, which, upon a link-state change, broadcasts a configuration request message to retrieve a dynamic address assignment. In most instances, DHCP is able to reconfigure the client without the use of beaconing or the extended use of polling; the polling stops as soon as the DHCP server responds to the client's message. There is no need to rely on polling or beaconing to detect migration because the next link-state change would trigger DHCP to reconfigure the client.

While the media sensing method works well for DHCP, it does not provide adequate micro-mobility detection in wireless networks. Consider when a client roams between two overlapping APs belonging to two different subnets. From our experience, some of the WNICs we have experimented with do not trigger DHCP to verify its client address and reconfigure the client when it is necessary. From the WNIC's point of view, this is the correct behavior because it is agnostic to the IP-level topology. The WNIC's default behavior is to handle the common case where the client stays within the same subnet as she roams between two APs.

The absence of a set of well-defined, consistent media sensing capabilities across different network interface technologies and their implementations has reinforced our design decision to use beacons, which is a hardware agnostic approach for mobility detection. We should note that the tradeoff of increasing beaconing frequency for reducing mobility detection time should not be significant. For instance, we can send beacons (on the order of a few hundred bytes) at the rate of 1Hz, which translates to negligible overhead in an 11Mbps wireless network.

8.5 Security

A good question to ask when examining the design of any auto-configuration system is how well does the system enforce security in the face of malicious attacks. In this section, we will concentrate on security issues that affect the auto-configuration part of the PANS system. For a full discussion about security topics concerning the PANS protocol, please refer to [3]. We will assume these security features from the PANS protocol throughout our discussion:

- The key and any other relevant parameters can be downloaded securely from the Authorizer to the client during the authentication process.
- The client can use the full packet encryption feature provided in PANS to increase security.

The auto-configuration system uses beacons to trigger client host configuration. Hence, the beacon becomes the entry point for possible attacks against the auto-configuration mechanism. Below, we will illustrate two types of attacks against our system and suggest possible security measures to guard against them.

8.5.1 Denial of Service (DoS)

A malicious user may learn the beaconing frequency and jam or intercepts the beacons at the predicted rate. Without detecting the beacons, the clients are denied access to the public network.

While we cannot prevent all forms of DoS attacks (such as jamming the entire wireless channel), we should make the attack difficult and/or detectable so that the service provider is alert to such an attack. First, the beaconing intervals can be randomized so that the attacker must either try to jam the entire channel (in the wireless network) or intercept the beacons on the physical network. These measures increase the difficulty of the attack by increasing the attacker's exposure to the system, thus reducing the chances of that attack go unnoticed. Whether there is an attack or not, the system should implement a network monitoring mechanism to ensure that the public network is operating normally. As an example, receivers of the network monitoring system can be installed throughout the physical area of a wireless public network. These receivers will monitor the frequency and integrity of each beacon being broadcasted by the individual APs. Although a malicious attacker can fool a receiver by replaying short-range beacons towards it, the attacker must devise such a device and possibly leave traces of evidence about the attack.

8.5.2 Hijacking

An attacker can redirect a client's packet stream by sending a false beacon containing an illegitimate Authorizer and/or Verifier address. The client can guard against this by performing integrity checks and authentication for each beacon. However, such technique is very costly and should be avoided. As an alternative, the network can set up a pair of public and private keys. In this scheme, the client must authenticate the Authorizer upon connection by, for example, checking its certificate. Then the client obtains the public key from the trusted Authorizer after she successfully gains access to the network. Just as she migrates her connections to a Verifier server, the client will issue a challenge to the Verifier.

The Verifier must return the challenge encrypted with the network's private key. The client will authenticate the Verifier by decrypting the challenge with the network's public key to see if it matches the original challenge it had sent to the Verifier. As an added measure of security, the client should use full packet encryption as provided by the PANS protocol.

9 Related Work

We are aware of a considerable amount of on-going work in the areas of Internet protocol design that addresses pieces of functionality that the CHOICE network provides. Although CHOICE combines and covers a broad range of ideas in existing work, we will discuss the work most relevant to our authenticated network access system and to our dynamic host auto-configuration system. We point the interested reader to [3] where additional details and comparisons are provided.

In the area of providing authenticated access to users, the two layer-2 mechanisms described in the IEEE 802.11 standard [2] (a) MAC-level filtering, and (b) the wired equivalency protocol (WEP) are insufficient for deployment in a public wireless networks. MAC-level filtering is difficult to manage and doesn't scale well, and WEP lacks the necessary hardware support for large-scale key management on a per-user basis. Other hardware-centric proposals include [12], [30], and [14]. Of these the most recent and promising one is the IEEE 801.1X standards committee's port-based network access control proposal which carries out layer-2 authentication by carrying the Extensible Authentication Protocol (EAP) frame within the Ethernet frame [12]. However, all of these proposals address only one aspect in our system and they do not consider issues like accounting, service quality, and user mobility. This last point is particularly important and has been discussed in detail in this paper.

The only fully deployed and documented authenticated network access system that we are aware of is the SPINACH system developed as part of the MosquitoNet project at Stanford University [13]. The strengths of SPINACH are the innovative reuse of existing infrastructure with no requirement for additional software in the client. However, this advantage also limits its functionality to user authentication only. The CHOICE system requires client side software but because of this is able to incorporate service quality and mobility support in addition to authentication, privacy and security. Also, without IPsec [7] in place, the SPINACH system does not protect against hardware spoofing, whereas our system does.

As mentioned, there are some Internet protocols that can be combined to build part of our system. For example, IPsec authentication header (AH) [15], IPsec encapsulating security payload (ESP) [16] and IKE [17] can be used to solve the problem of privacy and security. However, the strength, power, and feature-richness of these protocols come at the cost of overhead that may be slightly too expensive for the average handheld wireless device. In CHOICE, we reduce the cost of bearing last-hop encryption by implementing a lightweight protocol to meet the specific needs of our service model. Where the need arises, clients can still use IPsec on CHOICE for strong protection of their individual end-to-end connections.

Moreover, IPsec couples user keys and security association tightly with IP level information. This directly impacts our goal of supporting roaming users whose IP address changes frequently. This point has been addressed in this paper where we have described a system that decouples key information from IP level information and consequently supports mobility with fast hand-offs.

In the area of supporting mobility, there is Mobile-IP (v4 and v6) [11], which employs a service discovery scheme based on ICMP router discovery. The method of service discovery is similar to ours except that our system does not provide a mechanism to probe the network for the target service. Service discovery protocols, such as Berkeley SDS [20] and MIT INS [19] can be used for locating and using some network services. However, these systems mainly address the problem of handling a large number of services in a highly dynamic environment, which is overkill for our application.

In the area of host configuration, DHCP is perhaps the most relevant piece of work [4]. Our system relies on it to configure the client's IP address. Although DHCP provides a set of configurable options field, we have defined a separate beaconing mechanism for our host configuration application. The primary reason for using a beaconing mechanism is to support fast mobility detection, dynamic failure recovery, and location information delivery.

CHOICE is designed with a specific set of user-centric requirements and tries to combine the strengths and features of the on-going efforts mentioned in this section to build a comprehensive system that is self-contained, hardware agnostic, and protocol agnostic. We have designed it so that the client software can be downloaded and installed on-site giving the service provider considerable flexibility in personalization.

10 Conclusion

The CHOICE network is a case study of computing and communications in public places. We have designed and deployed this network at a popular mall with the hope that it will provide us a research platform for studying how the general public actually uses such networks and the sorts of services they care about. We are unaware of any working, deployed and documented system that addresses all the issues we tackle in our network. In this paper we focus on the specific problem of managing nomadic users as they move between differently configured public and private networks. That this problem is real is confirmed by our experience in supporting corporate employees who have their own private wireless network. Our solution to the problem has many advantages. Specifically, (a) It supports dynamic configuration of client devices, without user intervention, as nomadic users roam between public and private networks. (b) It achieves high availability of network services, network scaling and load-balancing, and (c) it supports location services that are currently not available in other networks. In describing our solutions we make the case that achieving true device mobility without any user intervention requires that we resolve many issues beyond the ones being worked on within standards committees like the IETF and the IEEE. The existence of standards in device programming and access point programming can help us achieve our ultimate of seamless mobility.

Acknowledgement

We would like to acknowledge and thank several individuals who have helped develop the CHOICE network. In particular, Anand Balachandran and Srinivasan Venkatachary are two of the original designers and implementers of PANS. Stephen Dahl helped us deploy the network at the Crossroads Mall; Pierre De Vries handled the legal formalities and helped us with usability issues while being our liaison with the product groups. Paul Hoeffer designed our web interaction. We also thank Dave Andersen of MIT, Prof. Dave Johnson of Rice University, and Prof. Mary Baker of Stanford University for the well appreciated constructive discussions.

References

[1] ITU-R Rec. M. 1225, "Guidelines for Evaluation of Radio Transmission Technologies for IMT-2000," 1999.

[2] IEEE 802.11b/D3.0, "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification: High Speed Physical Layer (PHY) Extensions in the 2.4 GHz Band, " 1999.

[3] P. Bahl, A. Balachandran, and S. Venkatachary, "The CHOICE Network – Broadband Wireless Internet Access in Public Places," MSR-TR-2000, February 2000

[4] R. Droms, "Dynamic Host Configuration Protocol," *IETF RFC 2131*, March 1997, http://www.ietf.org/rfc/ rfc2131.txt

[5] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. de. Groot, "Address Allocation for Private Internets," *IETF RFC 1597*, March 1994, http://www.ietf.org/rfc/rfc1597.txt

[6] Active Server Pages: http://msdn.microsoft.com/ workshop/server/asp/ASPover.asp

[7] R. Atkinson, "Security Architecture for the Internet Protocol", *IETF RFC 2401*, November 1998, http://www.ietf.org/rfc/ rfc2401.txt

[8] MS Passport: http:/www.passport.com

[9] T. Elgamal, S. Cotter, and the Netscape Security Team, "Netscape Security: Open-standard Solutions for the Enterprise,1998",http://developer.netscape.com/docs/manuals/ security/scwp

[10] R. Braden, "Requirements for Internet Hosts Communication Layers, *IETF RFC 1122*, October 1989

[11] Internet drafts from the IETF Working Group, "IP Routing for Mobile and Wireless Hosts (Mobile IP)," http://www.ietf.org/html.charters/mobileip-charter.html

[12] *IEEE Draft P802.1x/D1,* "Port Based Network Access Control," September 1999

[13] G. Appenzeller, M. Roussopoulos, and M. Baker, "User-Friendly Access Control for Public Network Ports," *Proceedings of INFOCOM '99*, March 1999

[14] E. A. Napjus, "NetBar - Carnegie Mellon's Solution to Authenticated Access for Mobile Machines," CMU White Paper, http://www.net.cmu.edu/docs/arch/netbar.html

[15] S. Kent and R. Atkinson, "IP Authentication Header," *IETF RFC 2402,* Nov. 1998, http://www.ietf.org/rfc/ rfc2402.txt

[16] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)," *IETF RFC 2406,* November 1998, http://www.ietf.org/rfc/ rfc2406.txt

[17] D. Harkins, and D. Carrel, "The Internet Key Exchange (IKE)," *IETF RFC 2409*, November 1998, http://www.ietf.org/rfc/ rfc2409.txt

[18] Microsoft Virtual Private Networking (VPN) White Paper, http://www.microsoft.com/ntserver/commserv/ deployment/planguides/VPNSecurity.asp

[19] W. Adjie-Winoto, W., E. Schwartz, H. Balakrishnan,, and J. Lilley, "The Design and Implementation of an Intentional Naming System.," In *Proceedings ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, Dec. 1999), pp. 186-201.

[20] S. Czerwinski,, B. Zhao, T. Hodes, A. Joseph, and R. Katz, "An Architecture for a Secure Service Discovery Service," In *Proceedings of the ACM/IEEE MOBICOM* (Seattle, WA, Aug. 1999), 24-35

[21] Bahl, P., and Padmanabhan, V., "RADAR: An In Building RF-based User Location and Tracking System." In *Proc. IEEE INFOCOM* (Tel-Aviv, Israel, Mar. 2000).

[22] Want, R., Hopper, A., Falcao,V., and Gibbons, J., "The Active Badge Location System. ACM Trnsactions on Information Systems 101," (January 1992), 91-102

[23] Priyanth, N., Chakraborty, A., Balakrishnan, B., "The Cricket Location-Support System," In *Proc. ACM/IEEE MOBICOM 2000* (Boston, MA, Aug. 2000).

[24] Cheshire, S., Baker, M., "Internet Mobility 4x4." In *Proc. SIGCOMM 1996*, August 1996.

[25] Snoeren, A., Balakrishnan, B., "An End-to-End Approach to Host Mobility," In *Proc. ACM/IEEE MOBICOM 2000* (Boston, MA, Aug. 2000).

[26] Ramsdell, B., "S/MIME Version 3 Message Specification," IETF RFC 2633, June 1999, http://www.ietf.org/rfc/rfc2633.txt

[27] Hodes, T. D., Katz, R. H., "Composable Ad-hoc Location-based Services for Heterogeneous Mobile Clients," In *Proc. ACM/IEEE MOBICOM 1997* (Budapest, Hungary, Sept. 1997).

[28] Loat, C., Gross, G., Gommons, L., Vollbrecht, J., Spence, D., "Generic AAA Architecture," *IETF RFC 2903*, August, 2000.

[29] Schneirer, B., "Applied Cryptography," John Wiley & Sons, Inc., 1996.

[30] D. L. Wasley, "Authenticating Aperiodic Connections to the Campus Network," June 1996, http://www.ucop.edu/ irc/wp/ wp_Reports/wpr005/wpr005_Wasley.html

Puppeteer: Component-based Adaptation for Mobile Computing

Eyal de Lara[†], Dan S. Wallach[‡], and Willy Zwaenepoel[‡]

[†] Department of Electrical and Computer Engineering
[‡] Department of Computer Science
Rice University

Abstract

Puppeteer is a system for adapting component-based applications in mobile environments. Puppeteer takes advantage of the exported interfaces of these applications and the structured nature of the documents they manipulate to perform adaptation *without* modifying the applications. The system is structured in a modular fashion, allowing easy addition of new applications and adaptation policies.

Our initial prototype focuses on adaptation to limited bandwidth. It runs on Windows NT, and includes support for a variety of adaptation policies for Microsoft PowerPoint and Internet Explorer 5. We demonstrate that Puppeteer can support complex policies without any modification to the application and with little overhead. To the best of our knowledge, previous implementations of adaptations of this nature have relied on modifying the application.

1 Introduction

The need for application adaptation in mobile and wireless environments is well established [7, 12, 13, 20, 27, 32, 33]. On one hand, mobile environments are characterized by low and unstable resource availability. On the other hand, mobile users often want to access remote data using the same applications they use on their desktop machines. Unfortunately, many of these desktop applications require a rich and stable resource environment. They perform poorly when used on mobile clients, and require adaptation to provide acceptable levels of service. Many approaches to adaptation have been proposed before, and many taxonomies of adaptation are possible. We focus here on the types of adaptation policies as well as on where the adaptation is implemented.

Adaptation policies can be grouped into two types: *data* and *control*. Data adaptations transform the application's data. For instance, they transform the images in a document into a lower resolution format. Control adaptations modify the application's control flow (i.e., its behavior). For instance, a control adaptation could cause an application that otherwise returns control to the user only after an entire document is loaded to return control as soon as the first page is loaded.

Based on where the adaptation is implemented, we recognize a spectrum of possibilities with two extremes: system-based [21, 26] and application-based adaptation [14, 15, 18, 34]. With system-based adaptation, the system performs all adaptation by interposing itself between the application and the data; no changes are made to the application. With application-based adaptation, only the application is changed; the system is unaware of any adaptation. Application-based adaptation allows both data and control adaptation, while system-based adaptation is limited to data adaptation. System-based adaptation does not require modification of the applications, and provides centralized control, allowing the system to adapt several applications according to a system-wide policy.

In this paper, we present a novel approach to adaptation we call *component-based adaptation*. It enables application-specific control and data adaptation policies *without* requiring modifications to the application. It does so by using the exposed APIs of component-based applications and the structured nature of the documents they manipulate to implement application-specific control adaptation policies. Component-based adaptation attempts to bring together the benefits of system-based and application-based adaptation, namely to implement application-specific policies without modifying the applications. Since adaptation is done in the system, component-based adaptation retains the advantage of providing a centralized locus of control for adaptation of multiple applications.

Component-based adaptation enables policies that adapt by repeated use of *subsetting* and *versioning*. A subsetting policy creates a new virtual document consisting of a subset of the components of the original document (e.g., the first slide in a presentation). A versioning policy chooses among the multiple instantiations of a component (e.g., instances of an image with different resolution). The adaptation policies use the application's exposed API to extend the subset or to replace the version

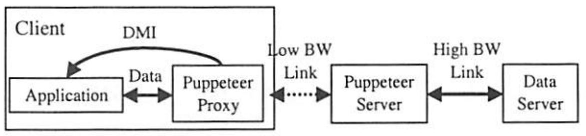


Figure 1: Overall system architecture.

of a component (e.g., load additional slides in a presentation or replace an image with one of higher fidelity). This iterative improvement is one of the key advantages of component-based adaptation over system-based adaptation.

Another approach that tries to strike a middle ground between system- and application-based adaptation is application-aware adaptation [6, 28]. Here, the system provides some common adaptation facilities, and serves as a centralized locus of control for the adaptation of all applications. The applications are modified to implement control adaptations and to perform calls to an adaptation API provided by the system. Component-based adaptation has similarities to application-aware adaptation in that both approaches delegate common adaptation tasks to the system. The approaches differ, however, in how control adaptation policies are implemented. In component-based adaptation, it is the applications that expose the interfaces, with the system invoking those interfaces to perform adaptation. The precise opposite occurs in application-aware adaptation where the applications are modified to call on the system's adaptation API. Component-based adaptation enables third parties to add new adaptation policies after the application has been released, while application-aware adaptation requires the application designer to foresee all necessary adaptations at the time the application is written.

Component-based adaptation is by nature restricted to component-based applications with exported APIs. While certainly a limitation, we observe that many desirable candidate applications for adaptation are already component-based, including the Microsoft Office Suite, Internet Explorer, Netscape Navigator, the KDE Office Suite, and Star Office. Recognizing the advantages of component-oriented software construction – independent of adaptation – we foresee an increasing number of applications being developed as components with exported APIs. Although traditionally associated with the Windows platform and with COM/DCOM technology, component-based technologies are becoming more common in the UNIX world as well, where the push for component-based technologies is led by the GNOME [1] and KDE [3] projects. A good example is KOffice [4], an open source productivity suite with powerful scripting capabilities. More recently, StarOffice [5] released version 5.2 of its popular cross-platform productivity suite, which implements a sophisticated object model that allows scripting by third party applications through a CORBA-based interface.

The more fundamental question about component-based adaptation is to what extent it can support the adaptation mechanisms that a customized application-based approach can achieve and with what performance. Furthermore, we wish to understand the scalability of component-based adaptation. Clearly, the system needs "drivers" for each application it wishes to support. For the concept to be scalable in terms of the number of applications it supports, the effort involved in writing an additional driver must be made small.

To address these questions, we have built a system we call Puppeteer. This paper describes the design of the Puppeteer system, its implementation on Windows NT, and our experience using this implementation to adapt two applications for low bandwidths, Microsoft PowerPoint (a presentation graphics system, hereafter "PowerPoint") and Internet Explorer 5 (a Web browser, hereafter "IE"). We demonstrate that Puppeteer can easily and efficiently support a number of desirable policies.

The rest of this paper is organized as follows. Section 2 presents the design of the Puppeteer system. Section 3 introduces the prototype implementation and the applications we use to evaluate it. Section 4 describes the experimental platform. Section 5 describes the documents we use in our experiments. Section 6 presents our experimental results. Section 7 discusses related work. Finally, Section 8 presents our conclusions.

2 Design

Figure 1 shows the four-tier Puppeteer system architecture. It consists of the application(s) to be adapted, the Puppeteer client proxy, the Puppeteer server proxy, and the data server. The application and data server are completely unmodified. The Puppeteer client proxy and server proxy work together to perform the adaptation.

The Puppeteer client proxy is in charge of executing the policies that adapt the applications. The Puppeteer server proxy is responsible for parsing documents, exposing their structure, and transcoding components as requested by the client proxy. The Puppeteer server proxy is assumed to have strong connectivity to the data server. In the most common scenario, it executes on the same machine as the data server. Data servers can be arbitrary repositories of data such as Web servers, file servers or databases.

2.1 Application Requirements

Puppeteer can adapt an application if it can uncover the component structure of its documents and if the appli-

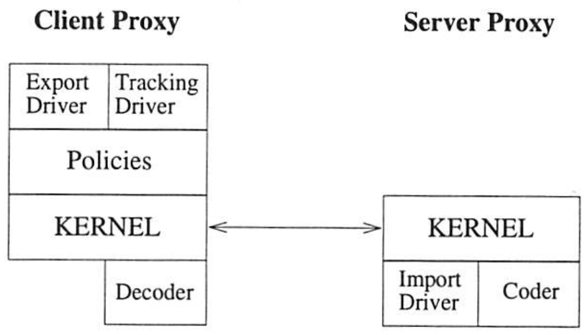


Figure 2: Internal Puppeteer architecture.

cation provides an API that enables Puppeteer to view and modify the data the application operates on. We refer to the latter feature as Data Manipulation Interface (*DMI*). Additionally, Puppeteer can benefit greatly from the ability to track the user's actions. We demonstrate next how Puppeteer implements adaptation once these requirements are met.

2.2 Puppeteer Architecture

The Puppeteer architecture consists of four types of modules: Kernel, Driver, Transcoder, and Policy (see Figure 2). The Kernel appears once in both the client and server Puppeteer proxy. A driver supports adaptation for a particular component type. A driver for a particular component type may call on a driver for another component type, if a component of the latter type is included in a component of the former type. At the top of this driver hierarchy sits the driver for a particular application (which itself is a component type). Drivers may execute both in the client and the server Puppeteer proxies, as may Transcoders which implement specific transformations on component types. Policies specify particular adaptation strategies and execute in the client Puppeteer proxy.

2.2.1 Kernel

The Kernel is a component-independent module that implements the Puppeteer protocol. The Kernel runs in both the client and server proxies and enables the transfer of document components. The Kernel does not have knowledge about the specifics of the documents being transmitted. It operates on a format-neutral description of the documents, which we refer to as the Puppeteer Intermediate Format (PIF). A PIF consists of a *skeleton* of *components*, each of which has a set of related *data items*. The skeleton captures the structure of the data used by the application. The skeleton has the form of a tree, with the root being the document, and the children being pages, slides or any other elements in the document. The skeleton is a multi-level data structure as components in any level can contain sub-components. The skeleton is component-independent, but components in the skeleton are component-specific. Component can have component-specific properties (e.g., slide title, image size) and one or more related data items that contain the component's native data.

When adapting a document, the Kernel first communicates the skeleton between the server and the client proxy. It then enables application policies to request a subset of the components and to specify transcoding filters to apply to the component's data. To improve performance, the Kernel batches requests for multiple components into a single message and supports asynchronous requests.

2.2.2 Drivers

For every component type it adapts, Puppeteer requires an import and an export driver. To implement complex policies, a tracking driver is also necessary. The import drivers parse the documents, extracting their component structure and converting them from their application-specific file formats to PIF.

In the common case where the application's file format is parsable, either because it is human readable (e.g., XML) or there is sufficient documentation to write a parser, Puppeteer can parse the file(s) directly to uncover the structure of the data. This results in good performance, and enables clients and server to run on different platforms (e.g., running the Puppeteer client proxy on Windows NT while running the Puppeteer server proxy on Linux).

When the application only exposes a DMI, but has an opaque file format, Puppeteer runs an instance of the application on the server, and uses the DMI to uncover the structure of the data, in some sense using the application as a parser. This configuration allows for a high degree of flexibility and makes porting applications to Puppeteer more straightforward, since Puppeteer need not understand the application's file format. It creates, however, more overhead on the server proxy, and requires both the client and server to run the environment of the application, which in most cases amounts to running the same operating system on both servers and clients.

Parsing at the server does not work well for documents that choose what data to fetch and display by executing a script, or by other dynamic mechanisms. Instead, import drivers for dynamic content run in the Puppeteer client proxy, and rely on an intercept mechanism that traces requests.

Regardless of whether the skeleton is built statically in the server proxy or dynamically in the client proxy, any changes to the skeleton are reflected by the Kernel at

both ends to maintain a consistent view of the skeleton.
Export drivers un-parse the PIF and update the application using the DMI interfaces exposed by the application. A minimal export driver has to support inserting new components into a running application.
Tracking drivers are necessary for many complex policies. A tracking driver tracks which components are being viewed by the user and intercepts load and save requests. Tracking drivers can be implemented using polling or event registration mechanisms.

2.2.3 Transcoders

Puppeteer makes extensive use of transcoding to perform transformations on component data. Transcoders include the conventional ones, such as compression and reducing image resolution. A novel transcoding mechanism is used to enable loading subsets of components. Each element of the PIF skeleton has a number of associated data items that, among other things, encode in a component-specific format the relationship between the component and its children. To load a subset of the children of a given node, it is sometimes necessary to modify the data items associated with the parent node to reflect the fact that we are only loading some of its children. In effect, by transcoding the parent node's data items, we create a new temporary component that consists only of a subset of the children of the original component.

2.2.4 Policies

Policies are modules that run on the client proxy and control the fetching of components. These policies traverse the skeleton, choosing what components to fetch and with what fidelity.
Puppeteer provides support for two types of policies: general-purpose policies that are independent of the component type being adapted (e.g., prefetching) and component-specific policies that use their knowledge about the component to drive the adaptation (e.g., fetch the first page only).
Typical policies choose components and fidelities based on available bandwidth and user-specified preferences (e.g., fetch all text first). Other policies track the user (e.g., fetch the PowerPoint slide that currently has the user's focus and prefetch subsequent slides in the presentation), or react to the way the user moves through the document (e.g., if the user skips pages, the policy can drop components it was fetching and focus the available bandwidth on fetching components that will be visible to the user).
Regardless of whether the decision to fetch a component is made by a general-purpose policy or by a component-specific one, the actual data transfer is performed by the Kernel, relieving the policy from the intricacies of communication.

2.3 The Adaptation Process

The adaptation process in Puppeteer is divided roughly into three stages: parsing the document to uncover the structure of the data, fetching the initially selected components at specific fidelity levels and supplying those to the application, and, if the policy so specifies, updating the application with newly fetched data.
When the user opens a (static) document, the Kernel on the Puppeteer server proxy instantiates an import driver for the appropriate document type. The import driver parses the document, extracts its skeleton and data, and generates a PIF. The Kernel then transfers the document's skeleton to the Puppeteer client proxy. The policies running on the client proxy ask the Kernel to fetch an initial set of components at a specified fidelity. This set of components is supplied to the application in return to its open call. The application, believing that it has finished loading the document, returns control to the user.
Meanwhile, Puppeteer knows that only a fraction of the document has been loaded. The policies in the client proxy now decide what further components or version of components to fetch. They instruct the Kernel to do so, and then the client proxy uses the DMI to feed those newly fetched components to the application.

3 Prototype

We use PowerPoint and IE as the initial applications for our prototype. Besides being widely popular, these two applications comply with the requirements for DMI, parsable file formats, and tracking mechanism from Section 2.1. Furthermore, PowerPoint and IE have radically different DMIs. By supporting both, we are more likely to accurately design the interfaces between the Puppeteer Kernel and the component-specific aspects of the system. Next, we discuss the design of the drivers, transcoders, and policies that we have implemented to adapt these two applications. Table 1 shows the code line counts for the various modules.

3.1 Drivers

3.1.1 Import Drivers

PowerPoint 2000 supports two native file formats: the traditional binary format based on OLE archives [22, 23], and a new XML-based format [24]. We choose to base the PowerPoint import drivers on the XML representation because it contains roughly the same informa-

Module		Code Lines
Kernel		8600
PPT	Import Driver	1114
	Export Driver	807
	Track Driver	112
	Transcoders	392
	Policies	287
	Total	2712
IE	Import Driver	314
	Export Driver	347
	Track Driver	65
	Transcoders	149
	Policies	334
	Total	1209

Table 1: Code line counts for Kernel, PowerPoint (*PPT*) and IE modules .

tion as the binary format, and the human readable nature of XML makes it easier to parse and manipulate the document. We have implemented import drivers for the following component types: PowerPoint, Slide, Images, Sound, Embedded Objects.
For IE, while HTML is straightforward to parse, the introduction of JavaScript in DHTML [16] has allowed for documents whose structure can change dynamically. For DHTML, the import driver intercepts URL requests, allowing it to dynamically add new images and components to a Web page's skeleton (see Section 2.2.2). We have implemented import drivers for the following component types: IE, Images.

3.1.2 Export Drivers

PowerPoint and IE DMIs are based on the Component Object Model (COM) [8] and the Object Linking and Embedding (OLE) [9] standards. The interfaces they provide are reasonably well documented [25, 31] and have traditionally been used to extend the functionality of third-party applications.
The PowerPoint and IE DMIs provide excellent access to compose and modify internal data structures. To support the policies we have implemented for this paper, the PowerPoint export drivers includes support for opening and closing presentations, and for inserting slides, images and embedded objects. The IE export driver includes support for navigating to a URL and for reloading individual components of a page.
To update an object in IE, the IE export driver instructs IE to reload only the URL associated with the object. PowerPoint supports a cut-and-paste interface to update a presentation. To paste new components into an active PowerPoint presentation, *active*, the PowerPoint export driver creates a new PowerPoint presentation, *helper*, that consists only of the new components. The update process has two stages. In *Stage 1*, the driver instructs PowerPoint to load *helper*. In *Stage 2*, for every component in *helper*, the driver copies it to the clipboard, pastes it into *active*, and deletes any earlier version of the same component from *active*.

3.1.3 Tracking Drivers

PowerPoint's event notification mechanism is primitive and encompasses just a handful of large-granularity events like opening or closing of documents, making it inadequate for tracking the behavior of the user. The PowerPoint tracking driver relies, instead, on polling the DMI to determine the slide currently being displayed.
The IE tracking driver uses IE's rich event mechanism that allows third-party applications to register call-back functions for a wide range of events. The driver uses this interface to detect when the user types a URL, presses the back or forward buttons, clicks on a link, or moves the mouse over an image. The former events are used to instruct the Kernel to open a new HTML document, while the latter is used by policies to drive image fetching and fidelity refinement (*e.g.*, refine the image currently pointed by the mouse).

3.2 Transcoders

The above policies use the following transcoders:

1 **Slide selector**. Creates a virtual presentation consisting of specific slides.
2 **OLE selector**. Creates a new file that contains only a subset of selected embedded OLE objects (PowerPoint stores embedded OLE objects in single file).
3 **Progressive JPEG**. Converts GIF and JPEG images into Progressive JPEG and back to JPEG.
4 **GZIP compressor**. Compresses and uncompresses text and binary data using gzip.

3.3 Policies

This section presents some sample adaptation policies that illustrate the power of component-based adaptation. These policies would be difficult to implement in system-based adaptation, because they affect not only the data used by the application, but also its control flow. Such adaptation policies have, to the best of our knowledge, only been implemented by modifying the application. In Puppeteer, however, they are implemented by using the external APIs. As will be demonstrated in Section 6 these policies also result in significant benefit under limited bandwidth conditions.

1 **PowerPoint: First slide**. Fetch only the components of the first slide at their highest fidelity, and

return control to the user. Fetch the rest of the presentation in the background.

2 **PowerPoint: Prefetch text**. Fetch all slides, but leave out any images and embedded objects. Monitor the user and fetch images and embedded objects of the slide that has the focus.

3 **IE: Incremental rendering**. Convert all GIF and JPEG images in a HTML page into Progressive JPEG. Load only the first 1/7 of the image, before returning control to user. Refetch with progressively higher fidelity the image pointed by the mouse.

3.4 Adding New Functionality

To adapt a new application with Puppeteer we need to implement drivers, policies, and transcoders for each new component type that is not currently supported by Puppeteer. For example, to enable MS Word we need to add drivers for the Word component type, but we can reuse the drivers and transcoders for the image and embedded object component types that we have implemented for PowerPoint (see Table 1).

While the effort in adding new applications and new policies is limited by the modular design of Puppeteer, the lack of standard DMIs, event models, and file formats requires new drivers to be written. Designing such standard interfaces is part of our ongoing research.

4 Experimental Environment

Our experimental platform consists of two Pentium III 500 MHz machines running Windows NT 4.0 that communicate via a third PC running the DummyNet network simulator [30]. This setup allows us to control the bandwidth between client and server to emulate various network technologies. For each application, we use three different bandwidths: one at which the application is network-bound, one at which it is CPU-bound, and one in-between.

All our experiments access data stored on an Apache 1.3 Web server. For the experiments where we measure the latency of loading the documents using the native application, Apache is the only process running on the server. For the Puppeteer experiments, the Apache server and Puppeteer server proxy run on the same machine.

5 Data Sets

We select the set of PowerPoint documents used in our experiments from a collection of Microsoft Office documents that we characterized earlier [11]. The full collection includes 2,167 documents downloaded from 334 Web sites with sizes ranging from 20 KB to 21 MB. We obtain our HTML documents by re-executing the traces of Web client accesses collected and characterized by Cunha *et al.* [10]. These traces include accesses from two user groups made during a period of 7 months from November 1994 through May 1995. These traces have 46,830 unique URLs corresponding to 3,026 Web sites. For every URL that we are able to access (many pages had either disappeared or were corrupted), we download the HTML file and any images referenced by them. We do not download any documents linked from these pages. In this manner we acquire 3,796 HTML files and 15,329 images, comprising 89 MB of data downloaded from 1,009 sites. Documents range in size from a few bytes to 773 KB, including images.

Because these data sets are so large, transmitting them at low bandwidth takes prohibitively long. We therefore run our experiments on just 92 PowerPoint documents and 182 HTML documents. For those subsets, the longest experiment requires 138 minutes for PowerPoint, and 55 minutes for HTML. For completeness, however, we run one test over the full sets of both document types over a high-bandwidth network, verifying that our selected documents and the full document sets produce similar results.

For our PowerPoint experiments, we select 92 documents by sorting all documents larger that 32 KB into buckets with sizes increasing by powers of 2. We then randomly select 10 documents from each bucket. The largest bucket, consisting of documents with sizes greater than 16 MB has only 2 documents. Thus, our experimental set has $9 \times 10 + 2 = 92$ members.

For our IE experiments, we select 182 HTML documents from the downloaded set by sorting all documents larger than 4 KB into buckets with sizes increasing by powers of 2. We then randomly select 25 documents from each bucket. The largest bucket, consisting of documents with sizes greater than 512 KB has only 7 documents. Thus, our experimental set has $7 \times 25 + 7 = 182$ members.

6 Experimental Results

The fundamental question that we want to answer in this section is how much overhead we pay for doing the adaptation outside of the application, as opposed to by modifying the application. To answer this question in a definitive way, we would need to modify the original applications to add the adaptation behavior that we achieve with Puppeteer, and compare the resulting performance to the performance of the applications running with Puppeteer. This is not possible, since we do not have access to the source code of the applications. Instead, we present some experiments to measure the various factors contributing to the Puppeteer overhead.

This overhead consists of two elements: a one-time initial cost and a continuing cost. The one-time initial cost consists of the CPU time to parse the document to extract its PIF and the network time to transmit the skeleton and some additional control information. Continuing costs come from the overhead of the various DMI commands used to control the application. We assume that other costs, such as network transmission, transcoding, and rendering of application data are similar for both implementations.

The remainder of this section is organized as follows. First, we measure the one-time initial adaptation costs of Puppeteer. Second, we measure the continuing adaptation costs. Finally, we present several examples of policies that significantly reduce user-perceived latency.

6.1 Initial Adaptation Costs

To determine the one-time initial costs, we compare the latency of loading PowerPoint and HTML documents in their entirety using the native application (*PPT.native, IE.native*) and the application with Puppeteer support (*PPT.full, IE.full*). In the latter configuration, Puppeteer loads the document's skeleton and all its components at their highest fidelity. This policy represents the worst possible case as it incurs the overhead of parsing the document to obtain the PIF but does not benefit from any adaptation.

Figures 3 and 4 show the percentage overhead of PPT.full and IE.full over PPT.native and IE.native for a variety of document sizes and bandwidths. Overall, the Puppeteer overhead for PowerPoint documents varies from 2% for large documents over 384 Kb/sec to 57% for small documents over 10 Mb/sec, and for HTML documents from 4.7% for large documents over 56 Kb/sec. to 305% for small document over 10 Mb/sec. These results show that, for large documents transmitted over medium to slow speed networks, where adaptation would normally be used, the initial adaptation costs of Puppeteer are small compared to the total document loading time.

Figure 5 plots the data breakdown for PowerPoint and HTML documents. We divide the data into application data and Puppeteer overhead, which we further decompose into data transmitted to fetch the skeleton (*skeleton*) and data transmitted to request components (*control*). This data confirms the results of Figures 3 and 4. The Puppeteer data overhead becomes less significant as document size increases. The data overhead varies for PowerPoint documents from 2.9% on large documents to 34% on small documents, and for HTML documents from 1.3% on large documents to 25% on small docu-

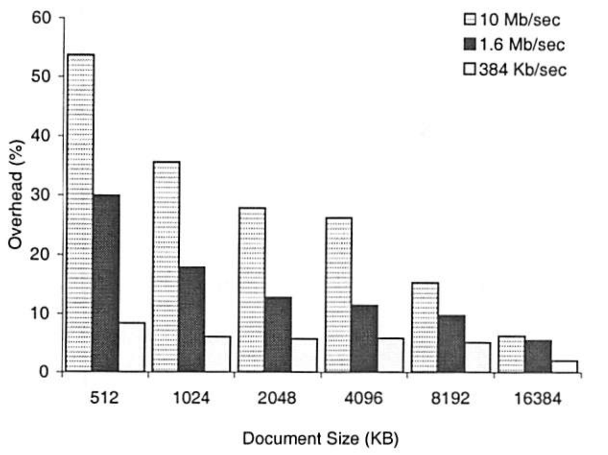


Figure 3: Percentage overhead of PPT.full over PPT.native for various document sizes and bandwidths.

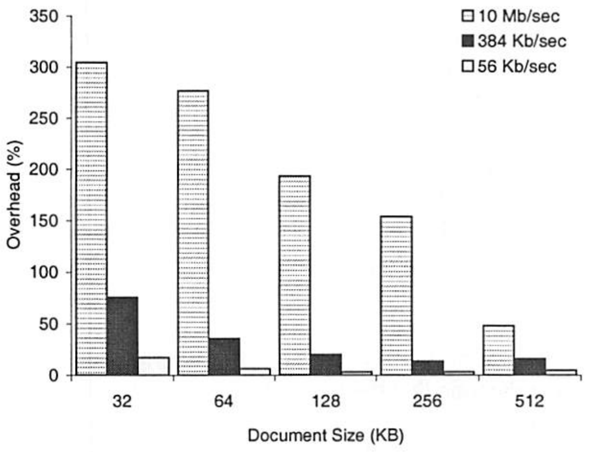


Figure 4: Percentage overhead of IE.full over IE.native for various document sizes and bandwidths.

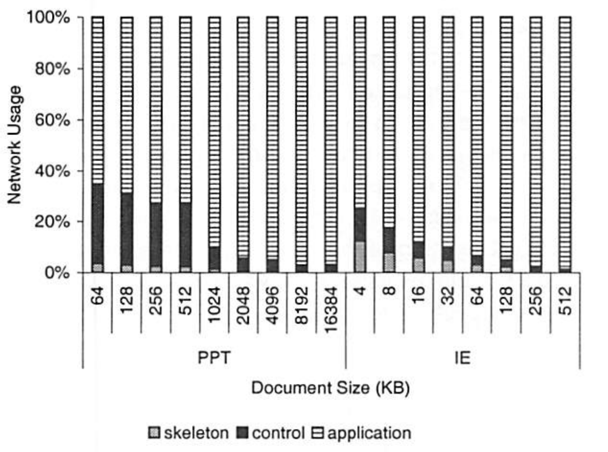


Figure 5: Data breakdowns for loading PowerPoint and HTML documents.

Operation		Cost (ms / component) Single Avg	Single Stdev	Additional Avg	Additional Stdev
Slide (PPT)	Stage 1	746	723	417	492
	Stage 2	148	96	113	99
Image (IE)	Synthetic	N/A	N/A	29	9
	DMI	33	19	32	12

Table 2: Continuing adaptation costs for PowerPoint (*PPT*) slides and IE images. The table shows the cost of executing OLE calls that append PowerPoint slides or upgrade the fidelity of IE images

ments.

6.2 Continuing Adaptation Costs

The continuing costs of adapting using the DMI are clearly dependent on the application and the policy. Our purpose is not to give a comprehensive analysis of DMI-related adaptation costs, but to show that they are small compared to the network and rendering times inherent in the application. We perform two experiments: loading and pasting newly fetched slides into a PowerPoint presentation, and replacing all the images of an HTML page with higher fidelity versions. To prevent network effects from affecting our measurements we make sure that the data is present locally at the client before we load it into the application.

We determine the PowerPoint DMI overhead by measuring the time that the PowerPoint export driver spends loading the new slides, *Stage 1*, and cutting and pasting, *Stage 2*, as described in section 3.1.2. We expect that an in-application approach to adaptation would have to perform *Stage 1*, but would not need to perform *Stage 2*.

For IE, we determine the DMI overhead for upgrading the images in two different ways: *DMI*, which uses the DMI to update the images; and *Synthetic*, which approximates an in-application adaptation approach. *Synthetic* measures the time to load and render previously generated pages that already contain the high fidelity images. *Synthetic* is not a perfect imitation of in-application adaptation, because it requires IE to re-load and parse the HTML portion of the page, which an in-application approach could dispense with. We avoid this problem by using only pages where the HTML content is very small (less than 5% of total page size), so that HTML parsing and rendering costs are minimal.

Table 2 shows the results of these experiments. For each policy, it shows the cost of updating a *single* component (i.e., one slide or one image) and the *additional* cost incurred by every extra component that is updated simultaneously. For PowerPoint, the table shows the time spent in *Stage 1* and *Stage 2*. For IE, the table shows the times for the *DMI* and *Synthetic* implementations.

The PowerPoint results show that the time spent cutting and pasting, *Stage 2*, is small compared to the time spent loading slides, *Stage 1*, which an in-application also has to carry out. Moreover, the time spent updating the application (*Stage 1* + *Stage 2*) is small compared to the network time. For example, the average network time to load a slide over the 384 Kb/sec network is 2994 milliseconds, with a standard deviation of 3943 milliseconds, while the average time for updating the application with a single slide is 994 milliseconds, with a standard deviation of 819 milliseconds.

The IE results show that the *DMI* implementation comes within 10% of *Synthetic*. Moreover, the image update times are small compared to the average network time. For instance, the average time to load an image over a 56 Kb/sec network is 565 milliseconds with a standard deviation of 635 milliseconds, compared to updating the application which takes on average 33 milliseconds with a standard deviation of 19 milliseconds.

The above results suggest that the cost of using DMI calls for adaptation is small, and that most of the time that it takes to add or upgrade a component is spent transferring the data over the network and loading it into the application. These two factors are expected to be similar whether we implement adaptation outside or inside the application.

6.3 Some Adaptation Policies

We conclude this section by presenting the results, as the end user would perceive them, of some of the Puppeteer adaptation policies we have implemented so far (see Section 3.3). These results also provide some indication of the circumstances under which these adaptations are profitable.

6.3.1 PowerPoint: Fetch First Slide and Text

In this experiment we measure the latency for a PowerPoint adaptation policy that loads only the first slide and the titles of all other slides of a PowerPoint presentation before it returns control to the user, and afterwards loads the remaining slides in the background. We also present results for an adaptation policy that, in addition, fetches all of the text in the PowerPoint document before returning control. With these adaptations, user-perceived latency is much reduced compared to the application policy of loading the entire document before returning control to the user.

The results of these experiments appear, under the labels *PPT.slide* and *PPT.slide+text*, respectively, in Figures 6, 7, and 8 for 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec network links. Figure 9 shows the data trans-

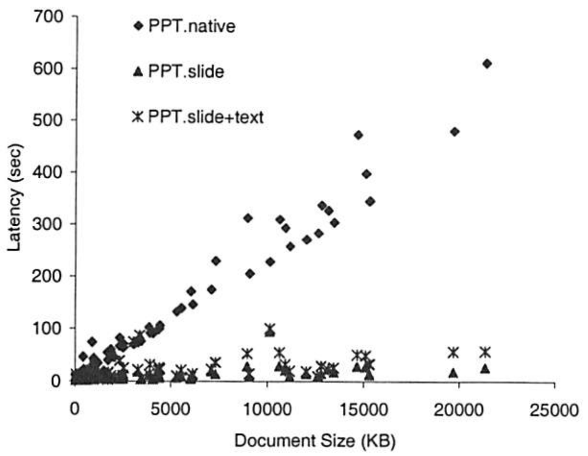


Figure 6: Load latency for PowerPoint documents at 384 Kb/sec. Shown are latencies for native PowerPoint (*PPT.native*), and Puppeteer runs for loading just the components of the first slide (*PPT.slide*), and loading, in addition, the text of all slides (*PPT.slide+text*).

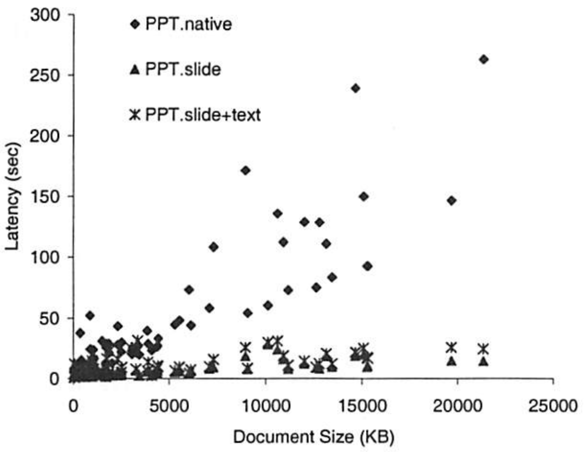


Figure 7: Load latency for PowerPoint documents at 1.6 Mb/sec.

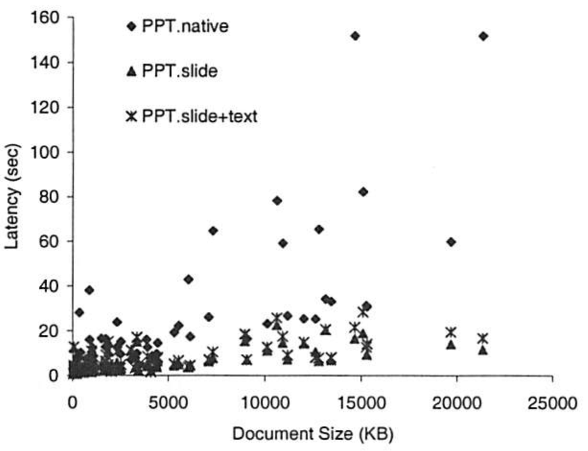


Figure 8: Load latency for PowerPoint documents at 10 Mb/sec.

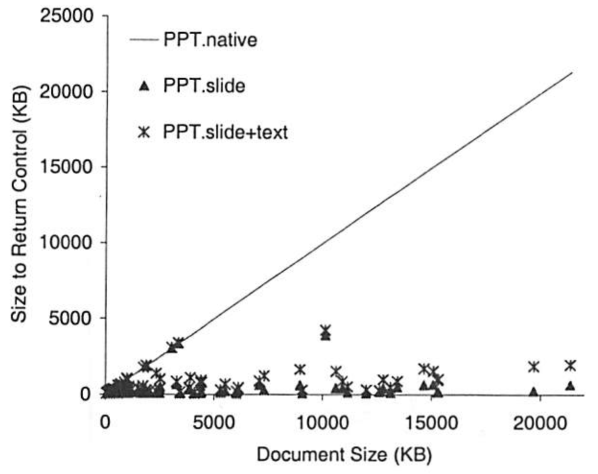


Figure 9: Data transfered to load PowerPoint documents.

fered in each of the three scenarios. For each document, the figures contain three vertically aligned points representing the latency or data measurements in three system configurations: native PowerPoint (*PPT.native*), Puppeteer loading only the components of the first slide and the titles of all other slides (*PPT.slide*), and Puppeteer loading in addition the text for all remaining slides (*PPT.slide+text*).

We expect that reduced network traffic would improve latency with the slower 384 Kb/sec and 1.6 Mb/sec networks. The savings over the 10 Mb/sec network come as a surprise. While Puppeteer achieves most of its savings on the 384 Kb/sec and 1.6 Mb/sec networks by reducing network traffic, the transmission times over the 10 Mb/sec are too small to account for the savings. The savings result, instead, from reducing the parsing and rendering time.

On average, *PPT.slide* achieves latency reductions of 86%, 78%, and 62% for documents larger than 1 MB on 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec networks, respectively. The data in Figure 9 also shows that, for large documents, it is possible to return control to the user after loading just a small fraction of the total document's data (about 4.5% for documents larger than 3 MB).

When comparing the data points of *PPT.slide+text* to *PPT.slide*, we see that the latency has moved up only slightly. The latency is still significantly lower than for *PPT.native*, achieving savings of, on average, 75%, 72%, and 54% for documents larger than 1 MB over 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec networks, respectively. Moreover, the increase in the amount of data transfered, especially for documents larger than 4 MB, is small, amounting to only an extra 6.4% above the data sent for the first slide. These results are consistent with our earlier findings [11] that text accounts for

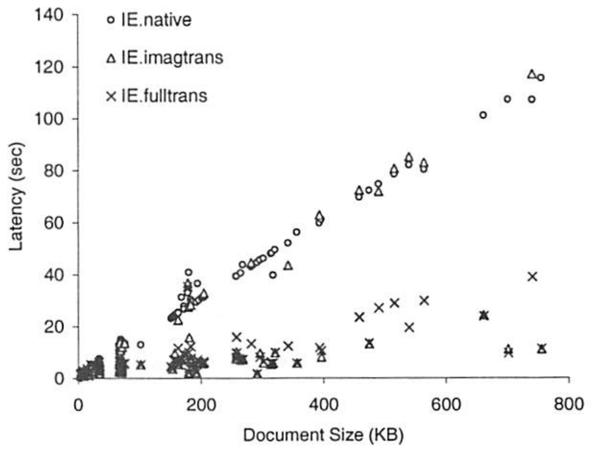


Figure 10: Load latency for HTML documents at 56 Kb/sec. Shown are latencies for native IE (*IE.native*), and Puppeteer runs that load only the first 1/7 bytes of transcoded images (*IE.imagtrans*), load transcoded images and text (*IE.fulltrans*).

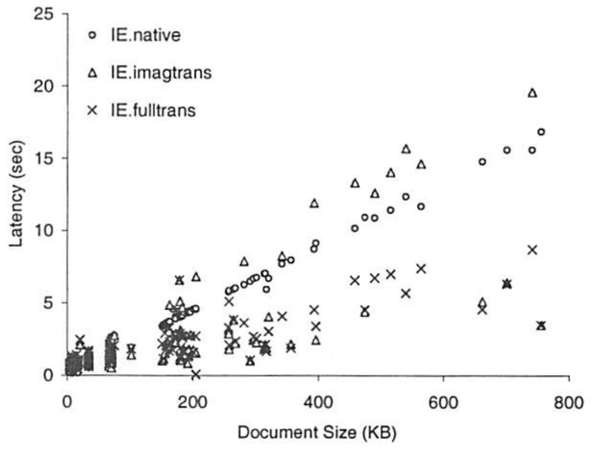


Figure 11: Load latency for HTML documents at 384 Kb/sec.

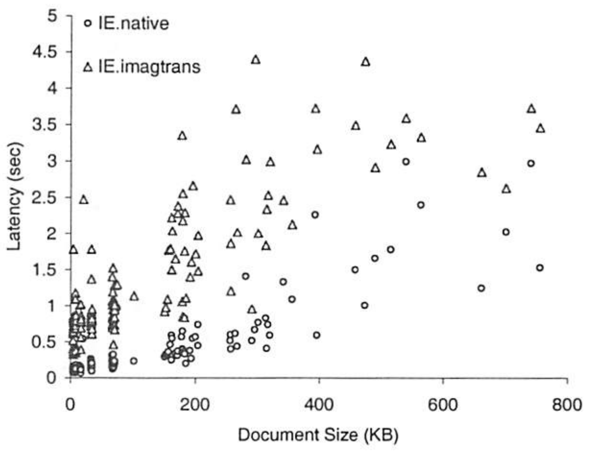


Figure 12: Load latency for HTML documents at 10 Mb/sec.

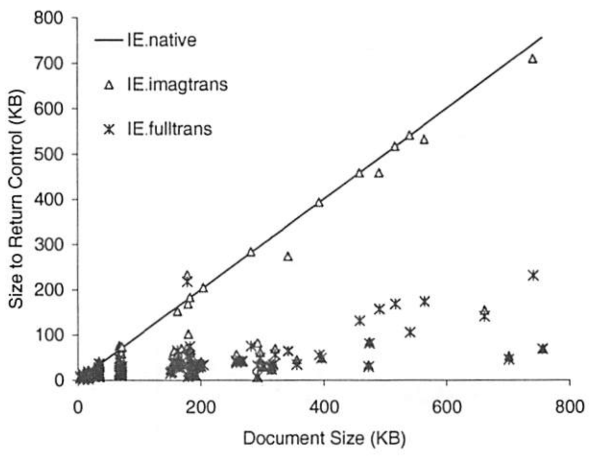


Figure 13: Data transfered to load HTML documents.

only a small fraction of the total data in large PowerPoint documents. These results suggest that text should be fetched in almost all situations and that the lazy fetching of components is more appropriate for the larger image and OLE embedded objects that appear in the documents.

Finally, an interesting characteristic of the figures is the large variation in user-perceived latency at high network speeds versus the alignment of data points into straight lines as the network speed decreases. The high variability at high network speeds results from the experiment being CPU-bound. Under these conditions, user-perceived latency is mostly dependent on the time that it takes PowerPoint to parse and render the presentation. For PowerPoint, this time is not only dependent on the size of the presentation, but is also a function of the number of components (such as slides, images, or embedded objects) in the presentation.

6.3.2 IE: JPEG Compression

In this experiment we explore the use of lossy JPEG compression and progressive JPEG technology to reduce user-perceived latency for HTML pages. Our goal is to reduce the time required to display a page by lowering the fidelity of some of the page's elements.

Our prototype converts, at run time, GIF and JPEG images embedded in an HTML document into progressive JPEG format[1] using the PBMPlus [29] and Independent JPEG Group [2] libraries. We then transfer only the first 1/7th of the resulting image's bytes. In the client we convert the low-fidelity progressive JPEG back into normal JPEG format and supply it to the browser as though

[1] A useful property of a progressive image format, such as progressive JPEG, is that any prefix of the file for an image results in a complete, albeit lower quality, rendering of the image. As the prefix increases in length and approaches the full image file, the image quality approaches its maximum.

it comprised the image at its highest fidelity. Finally, the prototype only transcodes images that are greater than a user-specified size threshold. The results reported in this paper reflect a threshold size of 8 KB, below which it becomes cheaper to simply transmit an image rather than run the transcoder.

Figures 10, 11, and 12 show the latency for loading the HTML documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec networks. Figure 13 shows the data transfered to load the documents. The figures show latencies for native IE (*IE.native*), and for Puppeteer runs that load only the first 1/7 bytes of transcoded images (*IE.imagtrans*), and load transcoded images and gzip-compressed text (*IE.fulltrans*).

IE.imagtrans shows that on 10 Mb/sec networks, transcoding is always detrimental to performance. In contrast, on 56 KB/sec and 384 KB/sec networks, Puppeteer achieves an average reduction in latency for documents larger than 128 KB of 59% and 35% for 56 KB/sec and 384 KB/sec, respectively. A closer examination reveals that roughly 2/3 of the documents see some latency reduction. The remaining 1/3 of the documents, those seeing little improvement from transcoding, are composed mostly of HTML text and have little or no image content. To reduce the latency of these documents we add gzip text compression to the prototype. The *IE.fulltrans* run shows that with image and text transcoding, Puppeteer achieves average reductions in latency for all documents larger than 128 KB, at 56 KB/sec and 384 KB/sec, of 76% and 50%, respectively.

Overall transcoding time takes between 11.5% to less than 1% of execution time. Moreover, since Puppeteer overlaps image transcoding with data transmission, the overall effect on execution time diminishes as network speed decreases.

As with PowerPoint, we notice in the figures for IE that for low bandwidths the data points tend to fall in a straight line, while for higher bandwidths the data points become more dispersed. The reason is the same as for PowerPoint: At high bandwidths the experiment becomes CPU-bound and governed by the time it takes IE to parse and render the page. For IE, parsing and rendering time depends on the content types in the HTML document.

7 Related Work

Much work has gone into supporting mobile clients [21] and into creating programming models that incorporate adaptation into the design of the application [17]. The project that most closely relates to Puppeteer is Odyssey [28], which splits the responsibility for adaptation between the application and the system. Puppeteer takes a similar approach, pushing common adaptation tasks into the system infrastructure and leaving the application-specific aspect of adaptation to application drivers. The main difference between the two systems lays in Puppeteer's use of existing run-time interfaces to adapt existing applications, whereas Odyssey requires applications to be modified to work with it.

Visual Proxies [34], an offspring of Odyssey, implements application-specific adaptation policies without modifying the application by using interposition between the X-server and the application. While this technique enables many adaptations that are possible with Puppeteer, it requires much more complicated application drivers.

The Dynamic Documents [19] system uses instrumentation of the Mosaic Web browser by Tcl scripts to set the policies for individual HTML documents. While Puppeteer uses the external interfaces provided by the application, Dynamic Documents uses an internal script interpreter in the browser.

8 Conclusions

We presented the design and measured the effectiveness of Puppeteer, a system for adapting component-based applications in mobile environments. Puppeteer implements adaptation by using the exposed APIs of component-based applications, enabling application-specific adaptation policies *without* requiring modifications to the application.

We described the architecture of Puppeteer and its implementation. The architecture allows for the modular addition of new applications, component types, transcoders, and policies. We demonstrated that complex policies, that traditionally require significant application modifications, can be implemented easily and efficiently in Puppeteer.

Puppeteer's reliance on application specific drivers to provide tailored adaptation raises the question of porting new application to the system. In our experience, the most time consuming part of porting an application is building the import driver that builds a PIF of the document by parsing the application specific file format. Once we had the necessary import and export drivers (the export drivers where considerably easier to implement), implementing policies proved surprisingly simple. In fact, most policies required less than a 50 lines of code.

With respect to standard file formats, the current trend towards XML-based formats has good promise. The only requirement is that components and their dependencies be made explicit. While we found the effort required to build export drivers to be modest, we are developing a set of standard APIs suitable for adaptation, including facilities for data manipulation and event registration.

References

[1] *GNOME*. http://www.gnome.org.

[2] Independent JPEG Group. http://www.ijg.org/.

[3] *KDE*. http://www.kde.org.

[4] *KOffice*. http://koffice.kde.org.

[5] *StarOffice*. http://www.stardivision.com.

[6] D. Andersen, D. Basal, D. Curtis, S. Srinivasan, and H. Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Octuber 2000.

[7] Rajive Bagrodia, Wesley W. Chu, Leonard Kleinrock, and Gerald Popek. Vision, issues, and architecture for nomadic computing. *IEEE Personal Communications*, 2(6):14–27, December 1995.

[8] Kraig Brockschmidt. *Inside OLE*. Microsoft Press, 1995.

[9] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[10] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW client-based traces. Technical Report TR-95-010, Boston University, April 1995.

[11] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Opportunities for bandwidth adaptation in Microsoft Office documents. In *Proceedings of the Fourth USENIX Windows Symposium*, Seattle, Washington, August 2000.

[12] Dan Duchamp. Issues in wireless mobile computing. In *Proceedings of Third Workshop on Workstation Operating Systems*, pages 1–7, Key Biscayne, Florida, April 1992.

[13] G H. Forman and J Zahorjan. The challenges of mobile computing. *IEEE Computer*, pages 38–47, April 1994.

[14] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. *Sigplan Notices*, 31(9):160–170, September 1996.

[15] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications*, 5(4):10–19, August 1998.

[16] D. Gardner. Beginner's guide to DHTML. http://wsabstract.com/howto/dhtmlguide.shtml.

[17] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 156–171, Copper Mountain Resort, Colorado, December 1995.

[18] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. In *Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom '96)*, Rye, New York, November 1996.

[19] M. Frans Kaashoek, Tom Pinckney, and Joshua A. Tauber. Dynamic documents: mobile wireless access to the WWW. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA '94)*, pages 179–184, Santa Cruz, California, December 1994. IEEE Computer Society.

[20] Randy H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[21] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[22] Microsoft Corporation, Redmond, Washington. *Microsoft Office 97 Drawing File Format*, 1997. MSDN Online, http://msdn.microsoft.com.

[23] Microsoft Corporation, Redmond, Washington. *Microsoft PowerPoint File Format*, 1997. MSDN Online, http://msdn.microsoft.com.

[24] Microsoft Corporation, Redmond, Washington. *Microsoft Office 2000 and HTML*, 1999. MSDN Online, http://msdn.microsoft.com.

[25] Microsoft Press. *Microsoft Office 2000 / Visual Basic Programmer's Guide*, 1999.

[26] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995.

[27] B. Noble and M. Satyanarayanan. A research status report on adaptation for mobile data access. In *SIGMOD Record*, volume 24, December 1995.

[28] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. *Operating Systems Review (ACM)*, 51(5):276–287, December 1997.

[29] Jeff Poskanzer. PBMPLUS. http://www.acme.com/software/pbmplus.

[30] L. Rizzo. DummyNet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, January 1997.

[31] Scott Roberts. *Programming Microsoft Internet Explorer 5*. Microsoft Press, 1999.

[32] M. Satyanarayanan. Hot topics: Mobile computing. *IEEE Computer*, 26(9):81–82, September 1993.

[33] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, Pennsylvania, May 1996.

[34] M. Satyanarayanan, J. Flinn, and K. R. Walker. Visual proxy: Exploiting OS customizations without application source code. *Operating Systems Review*, 33(3), July 1999.

Alpine: A User-Level Infrastructure for Network Protocol Development

David Ely, Stefan Savage, and David Wetherall

Department of Computer Science and Engineering
University of Washington, Seattle WA

Abstract

In traditional operating systems, modifying the network protocol code is a tedious and error-prone task, largely because the networking stack resides in the kernel. For this reason, among others, many have proposed moving the networking stack to user-level. Unfortunately, implementations of this design have never entered widespread use due to the impractical requirements they place on the user: either the kernel or applications must be modified; or code cannot be moved seamlessly between the user-level and kernel stacks. In this paper, we present Alpine, a user-level networking infrastructure free from these drawbacks. Alpine supports a FreeBSD networking stack on top of a Unix operating system. It is freely available as source code. In this paper, we discuss the challenges we faced in virtualizing the FreeBSD networking stack without compromising on kernel, networking stack, and application compatibility. We then show how Alpine is effective at easing the burden of debugging and testing protocol modifications or new network protocols. In our experience, Alpine can reduce the overhead of modifying a protocol from hours to minutes.

1 Introduction

The Internet enables a wide range of applications and supports clients with a wide range of connectivity, from low bandwidth mobile clients to clients with Gb/sec links; and yet there two protocols, UDP and TCP, that govern most of this communication. For this reason, many have proposed modifications and specializations to UDP or TCP to better serve the needs of applications. A literature search for proposed modifications quickly returns many results [3, 4, 5, 6, 10, 11, 13, 16, 18, 20, 23, 25, 28, 30, 31].

This work was funded by generous grants from NSF (DGE-9616736) and DARPA (F30602-00-2-0565). Correspondence concerning this paper may be sent to ely@cs.washington.edu.

Because the networking stack is traditionally part of the operating system, most of these modifications were developed and tested on "live" kernels, which has many drawbacks. Moving the networking stack into a user-level library for development gains the following advantages over developing protocols in the kernel:

- *Shortened revise/test cycle.* Kernel development includes an additional step in the revise/test cycle: a system reboot. This inconvenience increases the turnaround between revisions from a few seconds to a few minutes.
- *Easier debugging.* User-level development allows for easier source-level debugging.
- *Improved stability.* When developing protocols in a user-level environment, an unstable stack affects only the application using it and does not cause a system crash.

To provide these advantages to protocol developers, we have created Alpine (Application Level Protocol Infrastructure for Network Experimentation), a publically available practical tool that moves network protocols into a user-level library to facilitate development. The key distinction between our system and other user-level networking infrastructures is that we have been unwilling to compromise on compatibility with either existing application or kernel code. Once modifications have been developed and tested using Alpine, they are easily moved back into the kernel because Alpine and the kernel use the exact same source files. Furthermore, Alpine works with unmodified application binaries and requires no kernel modifications.

In the process of developing Alpine, we identified three fundamental requirements of virtualizing kernel services to user-level:

- *Virtualization of hardware and kernel routines.* Routines normally available to the kernel but are not available in user-space must be simulated. Hardware must also be virtualized because the actual hardware devices cannot be accessed from user-level.

- *Synchronization with kernel services.* Resources that are shared between the virtualized service and the kernel must be kept consistent without changing the kernel.
- *Transparent integration with applications.* The original semantics of applications that use the virtualized services must be preserved without changing their source code.

Our implementation section provides a more detailed discussion of how Alpine satisfies each of these requirements. The remainder of the paper is organized as follows. We first discuss how our infrastructure relates to previous work and how it differs. Then we present our design and implementation of Alpine in more detail. We then evaluate Alpine's success based on several factors, including performance and ease-of-use. Finally, we conclude with some comments on our experience of building this infrastructure and give directions for future work.

2 Previous Work

Many have proposed moving some or all of the network stack's functionality to user-space. There are essentially three motivations behind this user-level networking:

1. *Improved performance over the kernel's stack.* Work has been done to design high-speed access to device hardware for low-latency cluster processing[8, 32]. In contrast, Alpine is focused on normal applications that use normal networking APIs.
2. *Per-application specialization.* Many have shown that special kernel modifications or downloadable kernel code make application specialization of the networking stack possible [7, 9, 14, 15, 21, 24, 27, 29, 33]. Alpine supports specialization, but this is not its focus. Since our design constraints include requiring no kernel or application modifications, Alpine cannot achieve the high-performance of many of these systems.
3. *Simplified development.* Work has also been done to make kernel development easier. For example, [12] redesigned the kernel from scratch to allow user and kernel modules to be interchanged. Likewise, [17] simplifies development by separating the operating system kernel into encapsulated components, which can be interchanged or reused. In contrast, we have developed Alpine for an unmodified legacy operating system with unmodified application binaries.

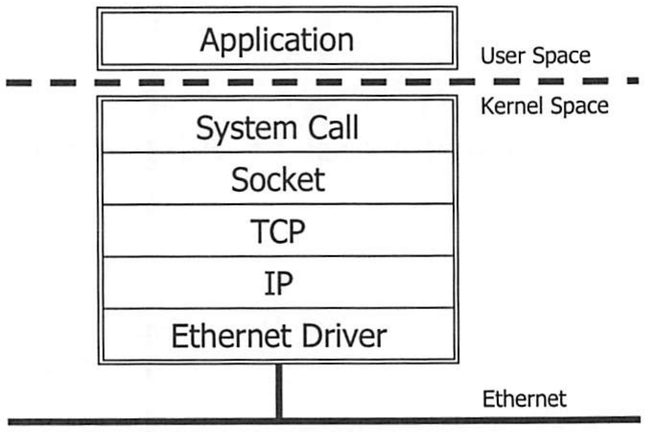


Figure 1: In a traditional network stack, applications interface with the network through a set of system calls (e.g., `socket`, `bind`, `send`), and all network processing is performed in the kernel.

While Alpine provides per-application specialization, its primary purpose is to simplify protocol development. Very little work has been done in this area. Alpine is the first user-level protocol development environment that runs on an unmodified legacy operating system and works with unmodified application binaries. We are focused on delivering an environment that makes modifying and developing networking protocols easier.

Some of our goals are shared by network simulators, which offer a convenient way to test an entire network on a single machine. Unlike Alpine, simulators are very rarely built using the kernel's networking stack. An exception to this is Entrapid [19], which is a simulator that allows a developer to simulate an entire network on a single machine using a modified FreeBSD networking stack. This system is intended more for developing higher level protocols because changes made in the modified stack might not be easily moved into the kernel. While a user can use our library to simulate multiple nodes in a network, Alpine is different from most network simulators [1, 22] because it is not confined to communicating only within a single machine or a single simulation environment.

3 Design

Alpine's primary goal is to be a practical platform for developing network protocols and protocol modifications. This goal lead us to four related design constraints:

- *No kernel changes.* Tools that require kernel modifications are difficult to deploy because of the general apprehension of installing unproven code in the kernel. Kernel modifications are often not portable to other versions of the same kernel, which limits the accessibility of tools that require them.

- *No application changes.* Requiring application changes more severely limits the usefulness of a development tool because *each* application must be modified. Also, the developer might be unfamiliar with the application source code or the application is available only in binary form.

- *No networking stack changes.* If the networking stack were altered, then moving protocol modifications back into the kernel would be difficult because the two stacks are built from separate source code.

- *No administrative oversight.* Administrative barriers, such as requiring an Alpine user to obtain a secondary IP address from their network provider, must be avoided.

In other words, we require Alpine to integrate transparently from the kernel, the application, and the programmer's perspective. This presents the dual challenge of virtualizing access to network and kernel resources while integrating this virtualized system into the native environment. These constraints limit how applications can interface with Alpine and how Alpine can interface with the operating system.

In the traditional Unix network design, a user application interfaces to the network through the socket interface. A socket is a unique communication channel between two hosts, which allow applications to connect to a remote computer, to send and receive data, and to listen for incoming connections. The socket API is a collection of system calls (e.g., `connect`, `sendmsg`, `recvmsg`, and `listen`). Figure 1 shows that the application interacts with the network exclusively through the socket API. The socket code calls into the transport layer (either UDP or TCP), which allows messages to be transmitted between hosts. After UDP or TCP has packaged the message, it sends it to the IP layer, which determines how to route the packet to the destination computer. After determining the appropriate route, IP sends the packet to the interface driver, which is responsible for actually putting the packet on the wire. Each layer includes additional information with the packet in the form of packet headers. In this architecture, the message crosses the dividing line between user-space and kernel-space very early (at the system call layer). This makes debugging the network stack difficult.

We propose to move this line much lower. In Alpine all of the packet processing and framing occurs inside of a user-level networking library. The fully-formed packet (including TCP/IP headers) is sent directly from user-space to the network interface. In our design, we have moved the barrier between user code and kernel code from the system call level to the interface driver. As Figure 2 shows, the unmodified socket layer and the TCP/IP layers have been moved into a library, which is responsible for sending and receiving packets and maintaining state about connections. Ideally, this library interacts with the interface (e.g., Ethernet card) directly, but for security reasons, the interface cannot be accessed without going through the kernel. For this reason, we wrote a software-only interface driver that does not control a hardware interface, but instead it sends packets using a raw socket and receives them using a packet capture library[1]. This system-independent device is termed a faux-ethernet driver because from IP's perspective, it is identical to a normal ethernet driver. To prevent the kernel's networking stack from reacting to packets destined for applications using Alpine, a firewall is installed that filters out Alpine's packets before they reach the kernel's stack.

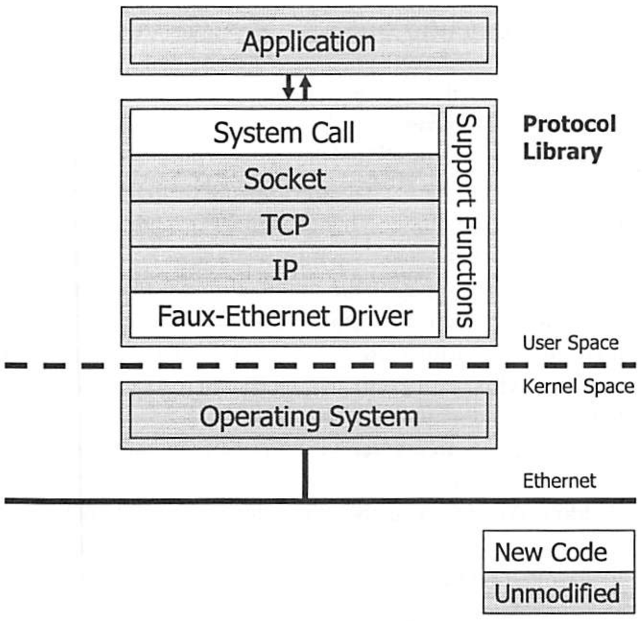


Figure 2: In Alpine the unmodified networking code is placed in a user-level library. An application is not modified because it interfaces with the library using the traditional socket system API.

At the application interface to the networking stack, we exported the same networking API as the kernel. This involved writing a pseudo-system call layer that replaced the traditional networking system calls with our own. We also provide some support code that emulates kernel functions upon which the networking stack relies. These mostly include synchronization functions and functions manipulating kernel data structures.

The following section discusses this design in more detail and attempts to classify the challenges we met when implementing Alpine.

[1]Using raw sockets and the packet capture library requires root privileges, but this is seen as only a minor inconvenience since modifying the kernel also requires root access. In section 6, we propose a solution to this shortcoming.

4 Implementation

We now discuss the implementation challenges of virtualizing network protocols with the design constraint that the kernel, the applications, and the networking stack remain unmodified. All of these challenges are either challenges in virtualization or challenges in integration. From the perspective of the networking stack we must *virtualize* kernel services; from the perspective of the application we must *virtualize* the system call interface to network protocols. We must also *integrate* this virtualized system into a native running environment. This involves managing shared resources such as a port space and a file descriptor table.

Achieving both service virtualization and seamless integration was the main technical challenge we overcame when building Alpine. The following are three fundamental requirements for virtualizing kernel services and integrating them into a running environment:

- Virtualization of hardware and kernel routines
- Synchronization with kernel services
- Transparent integration with applications

These three requirements are not unique to Alpine. They can, and should, be used as guidelines for virtualizing other kernel services. Organized according to these three requirements, our discussion of Alpine follows.

4.1 Virtualization of Hardware and Kernel Services

In Alpine an unmodified kernel networking stack runs inside a user-level library where it is not possible to supply all kernel facilities, such as direct access to hardware or fine grain timers. These facilities must be virtualized using only user-level services because the kernel cannot be modified to extend these services to user-level. In this section, we discuss the specific virtualization challenges that we met, including sending and receiving full-formed IP packets from user-level, simulating the boot process for the networking stack, and managing protocol timers.

Sending/receiving through the faux-ethernet interface

For security reasons, applications and user-level libraries cannot directly access a hardware interface but instead must go through the kernel. We use two different techniques for sending and receiving packets that are similar to accessing the interface directly but do not violate the security imposed by the kernel.

Sending. Sending a packet is straightforward. We chose to use a per-application raw socket that sends preformed IP packets to the operating system for transmission. A raw socket bypasses the transport and IP layer of processing and is sent directly to the hardware interface. The packets sent to a raw socket are identical to those sent to the interface, and there is no impact on the IP layer because it generates identical packets whether it is part of the kernel or part of Alpine.

Receiving. Receiving packets is more complicated. We use the libpcap packet capture library [2] to receive packets. This library enables a user application to receive copies of all packets that are received by a given interface. Other user-level protocol implementations have used this same approach to receive packets [24]. Although Alpine has access to all incoming and outgoing packets, it installs a Berkeley Packet Filter that discards packets destined for other applications. This limits the kernel resources needed to buffer packets that have been received.

Faux-Ethernet. We have encapsulated our methods of sending and receiving packets into a faux-ethernet device which presents itself to IP as any other interface. Although this interface has been named faux-ethernet, it is not specific to ethernets and can easily be adapted to other interfaces such as modems.[2] This faux-ethernet device is attached to the user-level stack during initialization. Because it is the only interface present in the user-level stack, all IP packets are sent through it.

Simulating interrupts

Packets can be sent synchronously, that is, when a user calls `send`, the packet can actually be placed on the wire before returning. But packets *arrive* asynchronously and cannot be processed in this fashion. The kernel receives an interrupt whenever a packet arrives, but this interrupt is not passed up to the application level. We could have modified the kernel to forward this interrupt to Alpine using signals, but this would have violated a design constraint. In order for Alpine to receive packets, it continually poll libpcap to check if a packet has arrived. This is done approximately once every millisecond by using SIGALRM to call an interrupt handler 1000 times per second. This could be done more often if we changed the granularity of the kernel's software timer, but polling for packets once every millisecond has been sufficient. Because we are polling for packets instead of receiving interrupts, there may be up to a 1ms delay between when a packet is received by the interface and when Alpine processes the packet.

Pseudo-kernel environment

Not surprisingly, the TCP/IP stack is not a completely separable part of the kernel. It relies on many features that are only available in the kernel, such as scheduling and certain

[2]It may be necessary to alter certain parameters, such as MTU, to match that of the actual machine interface.

memory allocation routines. The stack also relies on being properly initialized during system startup.

The FreeBSD kernel was modular enough to extract the networking stack without having to bring along a lot of additional code, but some kernel code not pertaining directly to the networking stack was imported for convenience. This includes code to manipulate certain system data structures, synchronization code, and code used for timeouts. The networking stack also relied on certain functions that could not be directly imported from the kernel.

Support Functions. As Figure 2 shows, we implemented a small set of support functions that emulate their counterparts in the kernel. These functions include several operations dealing with memory allocation.

Software Interrupts. The kernel's processing of incoming packets is asynchronous and driven by software interrupts. The interface driver and the protocol layer both use software interrupts to schedule packet processing routines. Like hardware interrupts, a priority level is assigned to each interrupt, and an interrupt service routine can only be interrupted by a higher priority interrupt. The kernel provides functions, such as `splnet` and `splhigh`, to raise the interrupt level and `splx` to restore a previous level. The networking stack often raises the interrupt level when executing critical regions of code to prevent shared data structures from being corrupted. Alpine includes implementations of these software interrupt functions. They are used primarily to prevent Alpine's SIGALRM handler from executing when the application is in a critical region of code. Beyond providing this support code, the second major issue was ensuring that all of the copies of system data structures are properly initialized.

Initialization. Alpine's initialization routine must be called before the user makes any calls into the library. Alpine supplies an `init` function that executes before any other library function is called. It initializes its own internal data structures as well as calling a modified version of the kernel's `main` function. The kernel's `main` function calls the various network initialization routines, such as `ip_init` and `tcp_init`. Finally, a set of dynamic ports is allocated to be used for sockets that are not explicitly bound to ports.

Timer management

An operating system performs many tasks. These include synchronous tasks such as flushing the file cache to disk or scheduling processes, and asynchronous tasks, such as handling user input or processing incoming packets. The networking stack uses `timeout` to handle synchronous events and `tsleep/wakeup` for asynchronous events. For Alpine's stack to function properly, we must correctly implement each of these functions and do so without affecting the semantics of the application.

Timeout. The kernel allows protocols such as IP and TCP to export two functions, `slowtimo` and `fasttimo`, which are called periodically. `Fasttimo` is called five times per second, while `slowtimo` is called only twice per second. TCP uses these functions to retransmit missing packets after a given interval and for delayed acknowledgements. Calling these functions five and two times per second is not difficult because Alpine is already using SIGALRM to poll for packets every millisecond. The `slowtimo` and `fasttimo` functions are instances of a more general problem. The kernel has a function, `timeout`, that allows an arbitrary function to be called after a specified number of clock ticks. This `timeout` function is used in multiple places in the TCP/IP stack, and we were able to import the kernel's `timeout` implementation with few modifications.

Tsleep and Wakeup. The function `tsleep` allows the caller to wait on a specific event until a timeout expires. This permits functions such as `recv` to wait until a packet arrives. The caller is restarted when the timeout expires or when `wakeup` is called on the appropriate event. For example, a socket can sleep on its incoming queue, and when a packet is appended to this queue, `wakeup` is called to restart the caller.

`Tsleep` and `wakeup` are not exposed to a user-level application but can only be used inside the kernel. Therefore, it was necessary to implement our own versions of `tsleep` and `wakeup` that preserve their original semantics. The kernel can suspend the caller process until the timeout expires or `wakeup` is called, but because the application and the user-level networking library run in the same process, doing so in Alpine would result in the process sleeping forever. This makes implementing `tsleep` and `wakeup` more difficult because Alpine must continue to run even if the application is blocked. In our simple implementation, `tsleep` busy waits on a global flag, which is set by `wakeup`. To reduce CPU utilization, `tsleep` sleeps for a few microseconds between checks of this global flag.

For an unmodified networking stack to run within Alpine, it was necessary to virtualize several kernel services using only user-level services. These virtualized services borrow heavily from the kernel implementations and are often more simple than the kernel version because the service is only used by a single process. Because the kernel and Alpine share certain state, including a port space and file descriptors, it is necessary to synchronize state with these kernel services. This is discussed in the next section.

4.2 Synchronization with Kernel Services

Alpine provides a service that is also provided by the kernel, and state shared with the kernel, such as ports and file descriptors, must be synchronized between the two stacks. The kernel assumes that it is the exclusive manager of this state, and due to our design constraints, Alpine is responsible for keeping this state consistent without violating the kernel's assumption. To minimize administrative burden,

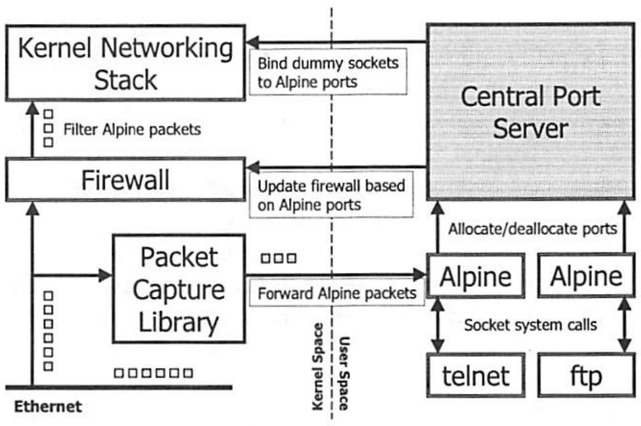


Figure 3: The role of Alpine's central port server is shown. The central server is responsible for allocating ports to each Alpine process. It uses a dummy socket to prevent the kernel from binding another socket to the port, and it uses a firewall to prevent the kernel's stack from reacting to packets sent to the bound port.

Alpine and the kernel's stack share an IP address. Thus, they share a port space that must be kept consistent. We found that having a central user-level process allocate ports to Alpine applications to be the best way to isolate faults and keep the two sets of ports synchronized. As section 3 mentions, Alpine interfaces with applications at the socket layer where Unix applications refer to sockets using file descriptors. Several system calls, including `open` and `close`, were overridden to synchronize Alpine's set of file descriptors (allocated to Alpine's sockets) with the kernel's file descriptors (allocated to files and pipes). Details of the methods we used to keep both the set of ports and file descriptors consistent follow.

Using a central server to mangage port allocation

Many conflicts between the kernel and Alpine can be avoided by allocating a separate IP address to Alpine. However, sometimes obtaining additional IP addresses is not feasible or desirable. In this case, the kernel and Alpine must share a single IP address and the port space that accompanies it. The user-level stack should not interfere with the port allocations of the kernel by re-allocating ports that the kernel has already allocated, and the kernel should not allocate ports that the user-library is using.

To solve both of these problems, each Alpine application allocates ports from a central user-level process. The central server's role in Alpine is shown in Figure 3 with its three specific duties listed below:

- A dummy socket is bound to each port that an Alpine application requests to prevent the kernel from allocating this port to another process.
- A firewall is installed that filters out packets destined for this port to prevent the kernel's stack from receiving and reacting to packets destined for an Alpine application. [3]
- After each Alpine process exits, the firewall is updated and the dummy socket closed for each port that the application was using, allowing other applications to bind to the port.

The primary reason for using a central server is to keep the firewall consistent with the set of ports that Alpine applications are using. Each Alpine application could update the firewall on its own, but this leads to two problems: 1) race conditions exist if multiple applications try to update the firewall concurrently and 2) if the process does not exit cleanly, the firewall may not be uninstalled properly, preventing other applications from using the blocked ports. Centralizing port allocation for Alpine processes is a reasonable way to ensure that the firewall remains consistent. Our approach is similar to that proposed in [29], which uses a "dedicated registry server" to handle connection setup and teardown.

Allocating Alpine file descriptors

For security reasons, applications in Unix cannot directly access a file on disk. Rather, they refer to open files using a file descriptor, which is merely an index into a per-process table of all open files. When a file is opened, a new file descriptor is created, and this file descriptor is passed to subsequent `read` and `write` calls to distinguish which file is being accessed. Because the socket API also uses file descriptors to distinguish between open sockets, we had to override all system calls that allocate or deallocate file descriptors.

Open and Close. A new file descriptor is created whenever `open` is called to open a file or pipe, or when `socket` is called to create a new socket. Because Alpine must know which file descriptors to allocate to its user-level sockets, it keeps track of the file descriptors being used by the kernel by overriding all system calls that create or delete file descriptors. This not only includes `open` and `socket`, which create a file descriptor, but also `close`, which deletes a file descriptor, and `dup`, which creates a copy of a file descriptor. As with the port space, the kernel and Alpine must share a set of file descriptors. Alpine cannot indiscriminately allocate file descriptors to the sockets that it creates because the kernel could allocate the same descriptor to a file in a future call to `open`. We solve this problem by opening a dummy file whenever a new user-level socket is created.

Whenever `socket` is called, Alpine assigns the same file descriptor to this socket as the kernel would, and then it opens the file "`/dev/null`," which prevents the kernel from allocating the chosen file descriptor to another file.

[3]Because the firewall runs on the same machine as the user-level stack, the packet capture library described previously is still able to receive all incoming packets.

This file is closed when the socket is closed. This approach allocates file descriptors identically to the kernel, preserving the original behavior of the application. A table is kept to distinguish our file descriptors from actual kernel file descriptors, enabling Alpine to correctly multiplex overloaded system calls such as `read`.

Although challenges of managing a shared port space and a shared file descriptor table may seem different, they essentially both involve keeping a shared namespace consistent. In fact, they are both solved using the same technique of attaching a false "name" (i.e. a dummy socket or a dummy open file) to a kernel resource to prevent the kernel from allocating it elsewhere.

4.3 Integration with Applications

For a development environment of Alpine's nature to be useful, it must work without modifying existing applications. For instance, having to rewrite application source code is unacceptable. Therefore, Alpine exports the traditional interface to network communication, the socket API. Furthermore, requiring recompiling or relinking of an application may seem acceptable, but this is sometimes inconvenient or impossible, which is why Alpine works with existing executable binaries. Exporting the socket API from Alpine requires manipulating the order in which the application is linked; by linking with the Alpine library before other libraries, Alpine's networking stack is used instead of the kernel's. To work with existing binaries, Alpine exploits dynamic linking; by loading Alpine's dynamic library before any other dynamic library, its networking stack is used instead of the kernel's. This technique cannot be used for applications that are statically linked. Fortunately, most applications are dynamically linked, especially those whose source code is unavailable.

There is an additional concern involved with preserving application semantics. We must ensure that none of the techniques Alpine uses to virtualize kernel services affects the semantics of the application. Two problems that could affect applications involve Alpine's `SIGALRM` handler which is used to perform periodic duties. First, we must allow the application to install a `SIGALRM` handler, and yet not allow the application to override Alpine's `SIGALRM` handler. Second, we must deal with the reentrancy issues introduced by having a signal handler that calls non-reentrant library routines.

Overriding socket system calls

Because we cannot modify the networking stack, we use a faux-ethernet device to send and receive packets. This is the interface that Alpine has with the operating system. A similar issue is at what level to interface with the user application. Applications interface with the network through the socket API, which is a set of system calls that allows an application to connect to another computer and send and receive data. As a design constraint we avoid application modifications, so having Alpine export a socket API is the only choice.

Send and Recv. System calls that are only used by sockets, such as `send`, `recv` and `connect`, were the simplest to implement. These system calls are replaced with our identically named functions, and as long as Alpine is loaded before libc, these socket system calls will be called instead of the kernel's.

Read and Write. In Unix, file descriptors are overloaded to refer to files, pipes, and sockets. With any of these types of "file", the user can call a certain set of overloaded functions including `read`, `write`, and `ioctl`. The operating system multiplexes calls to these functions into the appropriate file, pipe or socket function calls. For example, if read is called with a socket file descriptor then the system translates this into a call to `soreceive`. Therefore, we have to override these system calls and multiplex these calls between actual kernel files or Alpine sockets.

Select and Poll. Finally, parameters to functions such as `select` and `poll`, which determine if there is anything to "read" in a set of files or sockets, can include both file descriptors referring to files and to sockets. A timeout parameter associated with `select` and `poll` determines how long the operating system should wait for the "file" to become readable or writeable[4]. For instance, an application may block waiting either for a pipe to become readable or data to be received in a socket. `Select` will return when either the pipe or the socket becomes readable or when the timeout expires.

Alpine can determine locally if there is anything to read out of the socket buffers, but it must make a `select` call into the kernel to determine if there is anything to read out of the files. The timeout value cannot be passed through to the kernel because an incoming packet might cause a selecting socket to become readable. Thus, when an application is waiting on both a socket and a file, we poll (e.g., use a zero timeout) both the kernel file descriptors and the socket file descriptors until the timeout expires.

Transparent integration with existing binaries

Alpine can be used with unmodified application binaries by exploiting dynamic linking, which delays the binding of function calls until the application executes. The LD_PRELOAD environment variable allows the Alpine dynamic library to be loaded before any other library, which implies that Alpine's networking stack will be used instead of the kernel's. This enables Alpine to be used with any dynamically linked application.

[4]Libpcap's file descriptor cannot be passed directly to `select` because Alpine's SIGALRM handler would not run while the process was blocked, preventing packets from being retransmitted.

Application timers

Applications also use SIGALRM for timeouts and to perform periodic duties, but Unix only allows a single signal handler to be installed for each signal. We must not allow the application to replace Alpine's signal handler, however, we cannot prevent the application from using timers. To solve this problem, Alpine replaces many of the signal based system calls, such as `setsigaction` and `setitimer`, with its own implementations. Alpine records any SIGALRM handler that the application installs, but it does not change the actual handler for this signal. When the application schedules a SIGALRM to be delivered, the application signal handler is called from Alpine's signal handler after the application-specified delay. Because Alpine's signal handler is called at the highest possible frequency, it will always be able to call the application's signal handler at the correct time. However, if Alpine is executing a critical region of code, then this signal is delayed until the next clock tick. This is acceptable because the kernel can also delay delivery of signals for the same reason.

Non-reentrant library routines

Even though Alpine does not use threads, problems still arise with reentrancy because Alpine's SIGALRM handler can be called while the application is executing a non-reentrant library routine. For example, the signal handler should not call `malloc` if the application is updating a global data structure inside of `free`. To solve this problem, Alpine uses wrapper functions to place a lock around non-reentrant library routines, and its signal handler does not execute if the application is executing one of these routines. The application cannot call a non-reentrant library routine in an unsafe way because Alpine's signal handler always runs to completion.

To integrate with unmodified applications, Alpine is required to export the traditional socket interface to the network, and to ensure that the virtualization of kernel services has not altered the semantics of applications. We solved these two classes of challenges by exploiting properties of the linker, which allows Alpine to override any system call or library routine without modifying the application.

5 Experiences

Alpine has been fully implemented in the FreeBSD 3.3 operating system. However, very little of this code is specific to this version of FreeBSD, and most of it is portable to any Unix environment. It was successfully implemented without requiring modifications to the host operating system or applications using Alpine, and the same source files are used to build both Alpine and the kernel's networking stack. Alpine works with most applications, but it does not yet support applications that call `fork` because the fork produces a second networking stack. Open connections can be shared between the parent and child processes, which leads to problems in Alpine. For example, because the parent and child have their own copies of the networking stack, each will send acknowledgements to incoming TCP packets. In section 6, we discuss a way to extend Alpine to handle fork.

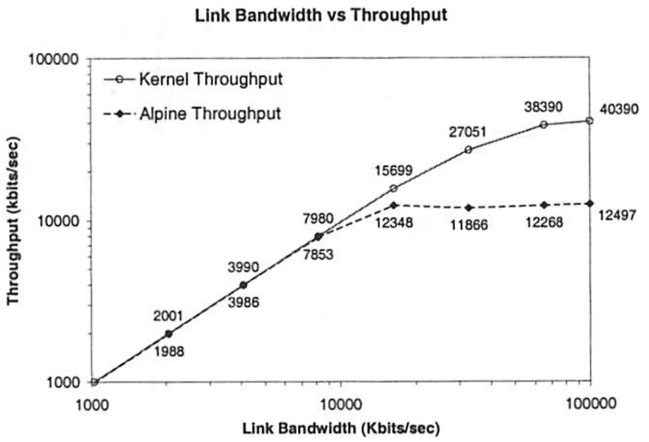


Figure 4: The throughput of both the kernel stack and Alpine are shown as the speed of the link increases. The extra copies, which are used by Alpine to maintain compatibility, limit it to a throughput of 12 Mbit/sec.

In the remainder of this section, we first compare Alpine's performance with the performance of the kernel's stack, and then we show possible uses of Alpine. While demonstrating ease-of-use quantitatively is difficult, we believe this section will enable the reader to understand the improvements Alpine makes over kernel development. The examples presented in this section are all related to TCP, but Alpine is certainly not limited to being used with TCP. It can be used to modify or test any transport level protocol.

5.1 Performance

Because Alpine is a user-level *development* infrastructure and is not intended as a replacement for the kernel's stack, its success does not depend on outperforming the kernel. However, Alpine must have reasonable performance in order to be a useful tool. Although certainly slower than the kernel's stack, Alpine can satisfy almost every application's bandwidth and latency requirements. Alpine cannot compete with the kernel's stack on the highest bandwidth links, although for link speeds up to 10 Mb/sec the two achieve similar performance. Latency in the local area is only slightly worse for the user-level stack.

Figure 4 shows how the kernel and Alpine performed as the link speed was varied. The test machine sent data as fast as possible to a second machine. Each machine was configured to use a third machine as a gateway, which used Dummynet to limit the bandwidth of the link [26]. All experiments were run on 200MHz Pentium-Pro PCs running

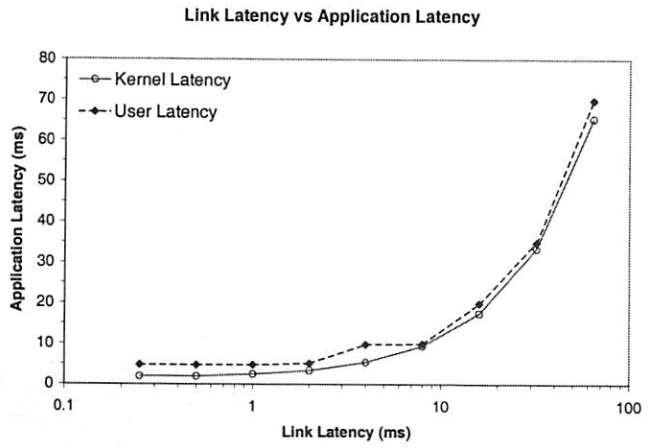


Figure 5: The application latency of the kernel stack and Alpine is shown. Due to additional overhead, the user-level stack is consistently 2.5 ms slower than the kernel network stack.

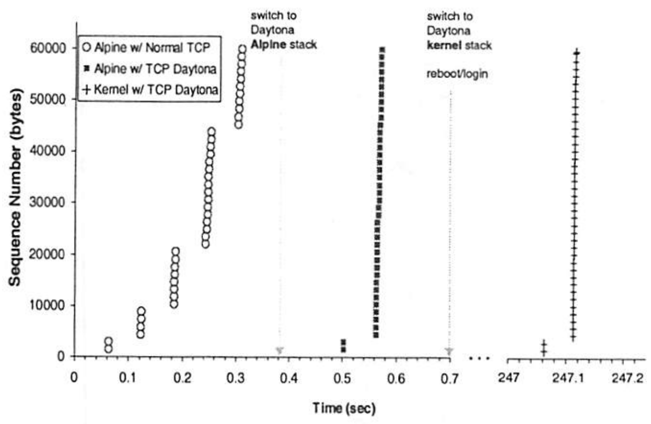


Figure 6: With Alpine very little time is needed to run an application with two different networking stacks, but it requires over four minutes to switch the kernel's networking stack.

FreeBSD 3.3. The two stacks are comparable up to link speeds of 10 Mbits/sec, where they start to diverge. While the kernel can achieve up to 40 Mbits/sec, the user-level stack can obtain at most 12 Mbits/sec. Less modest machines could achieve even higher performance.

The extra data copies Alpine needs to integrate seamlessly with applications and the unmodified kernel stack are responsible for this slowdown. In Alpine approximately five copies are necessary between when the application calls `send` and the packet is actually placed on the wire. Comparing this to the two or three copies the kernel needs, it is not surprising that Alpine cannot compete at higher bandwidths[5].

Figure 5 depicts how these extra copies and other overhead affect latency. In this experiment, one byte of data was sent to a remote computer, which immediately echoed the data. The link latency was varied from having no artificial link latency at .25 ms to having a 64 ms link latency. (In the local-area, .25 ms latencies are common, while in the wide-area, latencies of 30-60 ms are typical.) Alpine's latency was consistently 2.5 ms larger than the kernel latency, which is negligible for wide-area applications and is acceptable for most local-area applications.

We hope to improve the throughput and latency of Alpine, but the current design will always be slower than the kernel's stack. This is only a minor drawback because once protocol modifications have been tested in Alpine, they are easily moved back into the kernel where they can achieve higher performance.

[5]The additional copies required by Alpine occur at the interface to the application and the interface to the raw socket. The user buffer is copied into the kernel mbuf data structure which is shared by each layer of the protocol stack. Finally, the faux-ethernet driver copies the fully-formed packet out of the mbuf into a buffer, which is passed to the raw socket send.

5.2 Alpine Improves Protocol Development

By moving protocols into a user-level library, Alpine improves many aspects of protocol development. In this section, we try to give the reader an appreciation of these improvements.

Alpine shortens the revise/test cycle

With Alpine almost no time is needed between testing an application with two different networking stacks, and Figure 6 demonstrates this. The graph in Figure 6 is a time sequence plot where each mark represents a packet being received at the client. A 60 KB file is first downloaded using Alpine running a normal TCP stack, the same file is then downloaded using Alpine running TCP Daytona[6][28], and after a system reboot, the file is downloaded using the kernel stack running TCP Daytona. For all downloads, the same unmodified application was used.

The transition between the two Alpine stacks takes less than a second because only an environment variable must be changed to switch between the two stacks. However, switching to a different version of the kernel stack requires more than four minutes because the machine must be rebooted. While this example may seem extreme, it does demonstrate that time and effort are saved by eliminating the system reboot from the protocol development process.

Alpine improves protocol debugging

When an application uses Alpine, the networking stack runs in user-level. Thus, it can be examined and changed just as

[6]TCP Daytona is a set of modifications to TCP that allow the *receiver* to artificially override congestion control. This allows a client to gain an unfair share of network bandwidth without explicit help from the server.

any other part of the application source code. Any source-level debugger[7] can be used to set break points in the networking stack, step through untested modifications, or modify protocol state by changing protocol variables.

Figure 7 shows a screen shot of Alpine running in a graphical debugger. The large window in the foreground shows a telnet application stopped at the `tcp_output` function with the fields of the protocol control block (PCB) associated with this connection shown in the upper-half of the window. With Alpine, protocol state is easily displayed and modified. Tcpdump[2] runs at the bottom of the screen, showing packets that have already been transferred in the connection. The mail program and web browser running in the background are unaffected by the networking stack halted in the debugger because they are using the kernel's networking stack.

We have found that the power of user-level debuggers makes debugging protocol modifications much easier. Minor bugs that sometimes take hours to find in the kernel usually are found in only a few minutes when using Alpine.

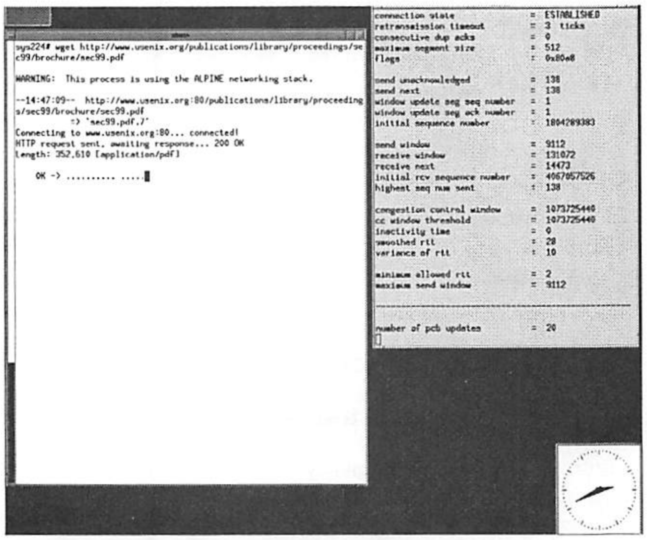
Alpine enables protocol instrumentation

To demonstrate that Alpine can be used in ways the kernel's stack cannot, we instrumented the Alpine networking stack to continuously display the fields of a TCP protocol control block (PCB). A TCP protocol control block (PCB) contains all of the relevant information pertaining to a single connection such as the initial sequence numbers and the size of the congestion window. Figure 8 shows this utility monitoring the `wget` utility while a file is being downloaded. When the PCB for this connection changes, the new PCB is automatically displayed in the window on the right. Achieving this task with the kernel's stack is more complicated because without modifying an application, it is difficult to instrument the kernel on an application-by-application basis. However, with Alpine this utility required only about an hour of programming, and it works with unmodified application binaries.

Alpine simplifies protocol modification

To test Alpine's usefulness when making protocol modifications, we made modifications of varying sizes to TCP. We had many choices for how to modify TCP. For example, we could have changed TCP to get better performance over wireless or satellite links. Instead, we chose to design and implement solutions to prevent the receiver-based TCP attacks, collectively named TCP Daytona, that Savage et. al.

[7]To seamlessly integrate with applications, we must prevent the SIGALRMs from being delivered while the application is stopped (e.g. at a breakpoint). This is easily done by using a debugger initialization file to install hooks that stop these signals when the application stops and resumes them when the application resumes. Alpine provides this initialization file for `gdb`, a popular debugger that is used as a back-end for most Unix-based debuggers.



```
sys224# wget http://www.usenix.org/publications/library/proceedings/se
c99/brochure/sec99.pdf

WARNING:  This process is using the ALPINE networking stack.

--14:47:09--  http://www.usenix.org:80/publications/library/proceeding
s/sec99/brochure/sec99.pdf
           => 'sec99.pdf.7'
Connecting to www.usenix.org:80... connected!
HTTP request sent, awaiting response... 200 OK
Length: 352,610 [application/pdf]

    OK -> .......... .....
```

```
connection state                =  ESTABLISHED
retransmission timeout          =  3 ticks
consecutive dup acks            =  0
maximum segment size            =  512
flags                           =  0x80e8

send unacknowledged             =  138
send next                       =  138
window update seg seq number    =  1
window update seg ack number    =  1
initial sequence number         =  1804289383

send window                     =  9112
receive window                  =  131072
receive next                    =  14473
initial rcv sequence number     =  4067057526
highest seq num sent            =  138

congestion control window       =  1073725440
cc window threshold             =  1073725440
inactivity time                 =  0
smoothed rtt                    =  28
variance of rtt                 =  10

minimum allowed rtt             =  2
maximum send window             =  9112
------------------------------------------------

number of pcb updates           =  20
```

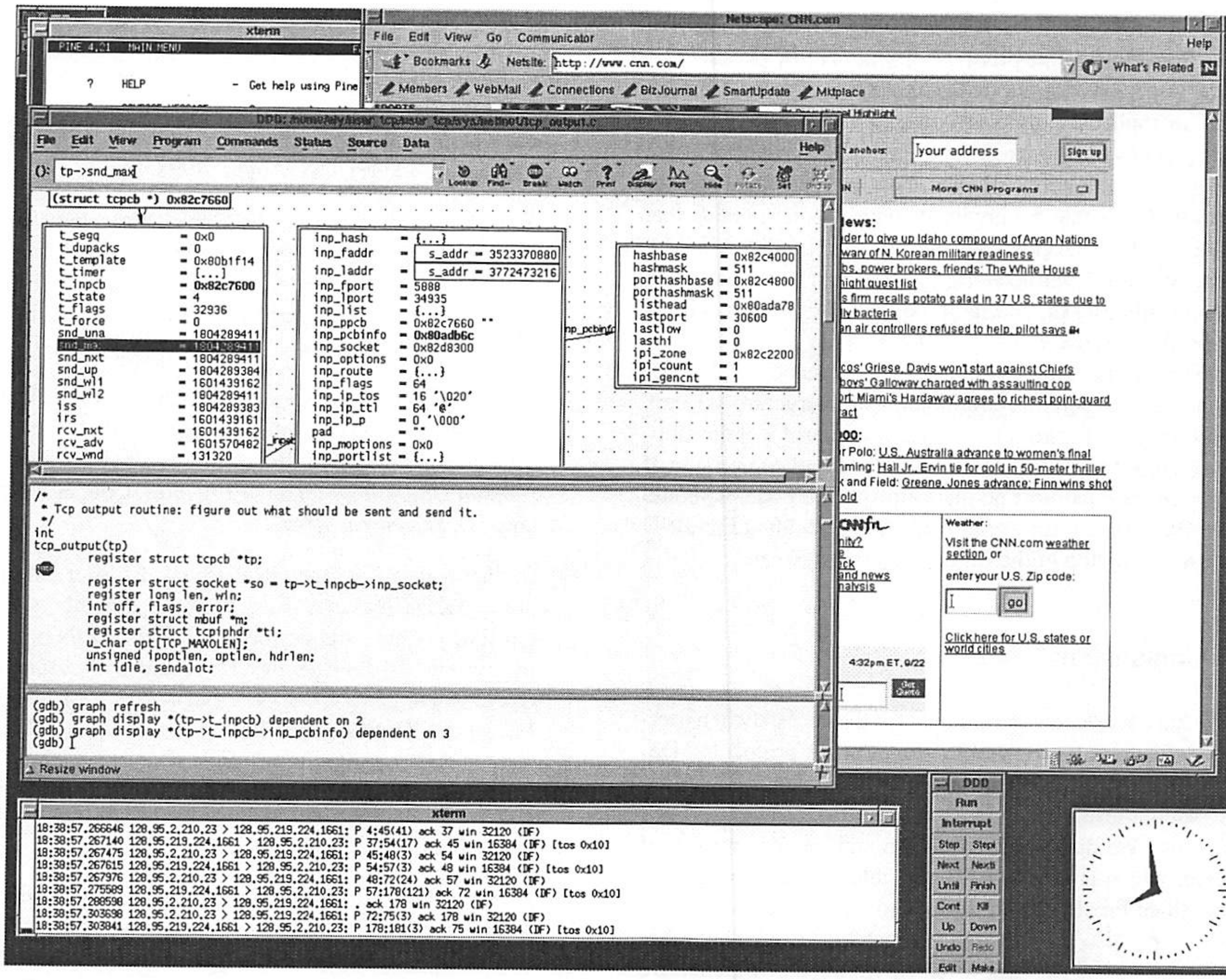
Figure 8: Alpine was used to extend the networking stack to display a connection's TCP PCB as it changes. The window on the right displays the values of fields in the PCB while the application on the left, which is using Alpine, downloads a file. Doing this in the kernel is much more difficult because it does not allow application-specific instrumentation. With the kernel's stack, the PCB must be displayed for all applications or none at all.

recently discovered [28]. TCP Daytona artificially forces congestion control to be overriden by manipulating *receiver* behavior, which allows the connection to gain an unfair share of network bandwidth. It is not the goal of this work to describe these attacks or discuss how we solved them. Instead, we provide insight into the benefits of using Alpine to implement and test our solutions.

Solutions to two of the attacks required modifications only to be made on the sender (e.g. server) and required approximately twenty lines of code to be added to the networking stack. We found Alpine to be a useful tool during the entire development process. Packet processing code is often written with attention paid to speed (i.e. avoiding procedure calls) instead of code modularity[8], which makes understanding the flow of control difficult. Using Alpine to step through the networking stack, we quickly found the proper place to implement these two solutions. Once our solutions were implemented, we were able to trace through the code to verify that the modifications behaved as expected, and we quickly discovered a bug, which was easily fixed. Once our solutions were sufficiently tested, they were moved into the kernel's stack without having to make any additional changes.

The solution to the third TCP Daytona attack required modifications to both the sender and the receiver (e.g. server

[8]For example, the two functions used to send and receive TCP packets, `tcp_input` and `tcp_output`, are over 1500 and 700 lines respectively. Understanding the flow of control in such functions is difficult without using a source-level debugger to step through the code.


xterm
PINE 4.21 MAIN MENU
? HELP - Get help using Pine
Netscape: CNN.com
File Edit View Go Communicator Help
Bookmarks Netsite: http://www.cnn.com/
What's Related
Members WebMail Connections BizJournal SmartUpdate Mktplace
your address
Sign up
More CNN Programs
Weather:
Visit the CNN.com weather section, or
enter your U.S. Zip code:
go
Click here for U.S. states or world cities
4:32pm ET, 9/22
DDD: /home/ely/user tcp/user tcp/sys/netinet/tcp_output.c
File Edit View Program Commands Status Source Data Help
Lookup Find Break Watch Print Display Plot Hide Rotate Set Undisp
Resize window
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
(): tp->snd_max

(struct tcpcb *) 0x82c7660

t_segq        = 0x0
t_dupacks     = 0
t_template    = 0x80b1f14
t_timer       = [...]
t_inpcb       = 0x82c7600
t_state       = 4
t_flags       = 32936
t_force       = 0
snd_una       = 1804289411
snd_max       = 1804289411
snd_nxt       = 1804289411
snd_up        = 1804289384
snd_wl1       = 1601439162
snd_wl2       = 1804289411
iss           = 1804289383
irs           = 1601439161
rcv_nxt       = 1601439162
rcv_adv       = 1601570482
rcv_wnd       = 131320

inp_hash      = {...}
inp_faddr     = s_addr = 3523370880
inp_laddr     = s_addr = 3772473216
inp_fport     = 5888
inp_lport     = 34935
inp_list      = {...}
inp_ppcb      = 0x82c7660 ""
inp_pcbinfo   = 0x80adb6c
inp_socket    = 0x82d8300
inp_options   = 0x0
inp_route     = {...}
inp_flags     = 64
inp_ip_tos    = 16 '\020'
inp_ip_ttl    = 64 '@'
inp_ip_p      = 0 '\000'
pad           = ""
inp_moptions  = 0x0
inp_portlist  = {...}

hashbase      = 0x82c4000
hashmask      = 511
porthashbase  = 0x82c4800
porthashmask  = 511
listhead      = 0x80ada78
lastport      = 30600
lastlow       = 0
lasthi        = 0
ipi_zone      = 0x82c2200
ipi_count     = 1
ipi_gencnt    = 1
```

```
/*
 * Tcp output routine: figure out what should be sent and send it.
 */
int
tcp_output(tp)
        register struct tcpcb *tp;

        register struct socket *so = tp->t_inpcb->inp_socket;
        register long len, win;
        int off, flags, error;
        register struct mbuf *m;
        register struct tcpiphdr *ti;
        u_char opt[TCP_MAXOLEN];
        unsigned ipoptlen, optlen, hdrlen;
        int idle, sendalot;
```

```
(gdb) graph refresh
(gdb) graph display *(tp->t_inpcb) dependent on 2
(gdb) graph display *(tp->t_inpcb->inp_pcbinfo) dependent on 3
(gdb) 
```

```
18:38:57.266646 128.95.2.210.23 > 128.95.219.224.1661: P 4:45(41) ack 37 win 32120 (DF)
18:38:57.267140 128.95.219.224.1661 > 128.95.2.210.23: P 37:54(17) ack 45 win 16384 (DF) [tos 0x10]
18:38:57.267475 128.95.2.210.23 > 128.95.219.224.1661: P 45:48(3) ack 54 win 32120 (DF)
18:38:57.267615 128.95.219.224.1661 > 128.95.2.210.23: P 54:57(3) ack 48 win 16384 (DF) [tos 0x10]
18:38:57.267976 128.95.2.210.23 > 128.95.219.224.1661: P 48:72(24) ack 57 win 32120 (DF)
18:38:57.275589 128.95.219.224.1661 > 128.95.2.210.23: P 57:178(121) ack 72 win 16384 (DF) [tos 0x10]
18:38:57.288598 128.95.2.210.23 > 128.95.219.224.1661: . ack 178 win 32120 (DF)
18:38:57.303698 128.95.2.210.23 > 128.95.219.224.1661: P 72:75(3) ack 178 win 32120 (DF)
18:38:57.303841 128.95.219.224.1661 > 128.95.2.210.23: P 178:181(3) ack 75 win 16384 (DF) [tos 0x10]
```

Figure 7: Alpine enables network protocols to be debugged using any source-level debugger. The application being debugged has stopped at a breakpoint in the `tcp_output` function, and the connection's TCP protocol control block (PCB) is being examined. Programs running in the background including a mail client and a web browser continue to use the kernel's networking stack.

and client), and required incorporating almost 100 lines of code into the networking stack. These changes were much more complex and required changing TCP's packet processing code in several places. Beyond the benefits listed previously, making these modifications exposed two other significant benefits of Alpine. First, Alpine allowed us to implement our solution in small increments. The high penalty of making a kernel modification leads many protocol developers to implement and test large pieces of code at once. This increases the number of bugs and probably is more costly in the long run. Our modifications were incrementally implemented and tested because of the absence of the kernel's high modification penalty. Second, Alpine enabled us to run both the sender and receiver concurrently on the same machine. We found this convenience to be very useful because we were able to trace through both stacks concurrently verifying that our modifications behaved as expected. Once this solution was completely tested, it was also moved into the kernel without any additional changes.

6 Future Work

Currently, Alpine's largest drawback is that root privileges are needed to use the infrastructure. (Opening a raw socket, capturing packets using libpcap, and installing a firewall all require root access.) A possible solution to this limitation is to move more functionality, specifically sending and receiving packets, into the central server, which is already used to manipulate the firewall. Once the central server is installed with root access, applications could use Alpine without any special privileges. The central server could also verify the source address of all packets being transmitted to prevent users from abusing the privilege of sending raw packets.

We also have not addressed the issue of what happens when an application forks. Because a forked process inherits its parent's open sockets, handling this issue is tricky. We plan to solve this problem by converting the socket system calls into RPCs that communicate with another process that contains only the Alpine networking stack. In this scenario, a separate user-level networking stack is not created when an application forks, and multiple applications can share a single Alpine stack.

Besides continuing the ongoing maintenance of Alpine, we also hope to port this development infrastructure to Linux and other Unix environments to facilitate protocol development on other platforms. Also, since the FreeBSD version of Alpine relies on almost no platform specific code, it should be easily ported to Linux, allowing an unmodified FreeBSD stack to run on top of the Linux operating system.

7 Conclusion

We have presented an argument for the necessity of a user-level infrastructure for developing network protocols. Developing outside the kernel has many advantages, including easy source-level debugging and quick turnaround between revisions. We discussed our design and implementation of Alpine, which is a publically available tool that enables an unmodified FreeBSD kernel stack to execute in a user-level library. We also presented guidelines for virtualizing other kernel services in a user-level environment. Finally, we showed that Alpine offers many improvements over traditional kernel protocol development. For more information about Alpine or to download the latest version of Alpine, please visit `http://alpine.cs.washington.edu/`.

References

[1] Ns network simulator. See `http://www-mash.cs.berkeley.edu/ns/`.

[2] Tcpdump home page. See `http://www.tcpdump.org/`.

[3] M. Allman. On the generation and use of TCP acknowledgments. *INFOCOM '98*, 28(5):4–21, October 1998.

[4] M. Allman. TCP byte counting refinements. *Computer Communications Review*, 29(3), July 1999.

[5] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. Request for Comments 2488, Internet Engineering Task Force, January 1999.

[6] H. Balakrishnan. *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*. PhD thesis, Computer Science Division, Univ. of California at Berkeley, Berkeley, CA, August 1998.

[7] A. Banerji, J. Tracey, and D. Cohn. Protected shared libraries-A new approach to modularity and sharing. In *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA*, pages 59–75, Berkeley, CA, USA, January 1997. USENIX.

[8] A. Basu, V. Buch, W. Vogels, and T. von Eicken. Unet: A user-level network interface for parallel and distributed computing. In *Proceedings 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO, December 1995.

[9] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain CO, December 1995.

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services: IETF RFC 2475, December 1998.

[11] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings, 1994 SIGCOMM Conference*, pages 24–35, London, UK, August 31st - September 2nd 1994.

[12] R. Draves and S. Cutshall. Unifying the user and kernel environments. Technical Report MSR-TR-97-10, Microsoft Research, March 1997.

[13] R. Durst, G. Miller, and E. Travis. TCP extensions for space communications. In *Proceedings, 1996 MobiComm Conference*, November 1996.

[14] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[15] M. Fiuczynski and B. Bershad. An extensible protocol architecture for application-specific networking. In *USENIX 1996 Annual Technical Conference*, San Diego, California, January 1996.

[16] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. Internet Draft,

Internet Engineering Task Force, February 1999. Work in progress.

[17] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: A substrate for os and language research. In *16th ACM Symposium on Operating Systems Principles*, October 1997.

[18] T. Henderson and R. Katz. Transport protocols for internet-compatible satellite networks. *IEEE Journal of Selected Areas in Communications*, 17(2):326–344, February 1999.

[19] X. Huang, R. Sharma, and S. Keshav. The Entrapid protocol development environment. In *INFOCOM '99*, New York, March 1999.

[20] V. Jacobson, B. Braden, and L. Zhang. TCP extension for high speed paths. *RFC 1185*, October 1990.

[21] M. F. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[22] S. Keshav. Real 5.0 network simulator. See `http://www.cs.cornell.edu/skeshav/real/overview.html`.

[23] L. Larzon, M. Degermark, and S. Pink. UDP lite for real time multimedia applications. Technical Report HPL-IRI-1999-001, HP Labroratories, April 1999.

[24] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. In *14th ACM Symposium on Operating Systems Principles*, December 1993. also CMU Technical Report CMU-CS-93-131.

[25] R. Ramanathan. TCP for high performance in hybrid fiber coaxial broad-band access networks. *IEEE/ACM Transactions on Networking*, 6(1):15–29, February 1998.

[26] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *Computer Communications Review*, 27(1):31–41, January 1997.

[27] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating system. In *Proceedings USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 39–69, 1992.

[28] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *Computer Communications Review*, 29(3), October 1999.

[29] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[30] F. Theodore, J. Touch, and W. Yue. The TIME-WAIT state in TCP and its effect on busy servers. In *INFOCOM '99*, New York, March 1999.

[31] V. Visweswaraiah and J. Heidemann. Improving restart of idle TCP connections. Technical Report 97-661, ISI, Marina del Ray, California, November 1997.

[32] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–267, Gold Coast, Australia, May 1992. ACM Press.

[33] D. Wallach, D. Engler, and M. F. Kaashoek. ASHs: application-specific handlers for high-performance messaging. *Computer Communications Review*, 26(4):40–52, October 1996.

Measuring Client-Perceived Response Times on the WWW

Ramakrishnan Rajamony and Mootaz Elnozahy
IBM Austin Research Lab, 11501 Burnet Road, Austin TX 78758
rajamony@us.ibm.com, mootaz@us.ibm.com

Abstract

The response time of a WWW service often plays an important role in its success or demise. From a user's perspective, the response time is the time elapsed from when a request is initiated at a client to the time that the response is fully loaded by the client. This paper presents a framework for accurately measuring the client-perceived response time in a WWW service. Our framework provides feedback to the service provider and eliminates the uncertainties that are common in existing methods. This feedback can be used to determine whether performance expectations are met, and whether additional resources (e.g. more powerful server or better network connection) are needed. The framework can also be used when a consolidator provides Web hosting service, in which case the framework provides quantitative measures to verify the consolidator's compliance to a specified Service Level Agreement. Our approach assumes the existing infrastructure of the Internet with its current technologies and protocols. No modification is necessary to existing browsers or servers, and we accommodate intermediate proxies that cache documents. The only requirement is to instrument the documents to be measured, which can be done automatically using a tool we provide.

1. Introduction

Businesses increasingly use the World Wide Web (WWW) to supply information such as news and movie reviews, and perform services such as stock and banking transactions for their customers. Responsive service plays a critical role in determining end-user satisfaction. In fact a customer who experiences a large delay after placing a request at a business's web server often switches to a competitor who provides faster service. Two factors contribute to the response time as perceived by a client: the network delay and the server-side latency (the time it takes to generate a response from the time the request reaches the web server). There are many established techniques for reducing the response time over the Web, including the use of powerful servers, reverse proxies at the server, Web caches, and clever load balancing among clustered or geographically dispersed servers. The services that Akamai [1], Digital Island [2], and Inktomi [3] provide are examples of how these techniques could work together.

Given the vital role played by response times in determining end-user satisfaction, businesses need to have quantitative information about the perceived response times of their services. This information guides the reaction of the business if the performance is below expectations. Server latency, i.e., the time it takes to service a request once it enters a web site, is an oft-used metric for infrastructure planning. However, server latency measures do not include network interactions and cannot represent user-perceived response times. For instance, server latency data sheds no light on potential problems within the network such as that caused by a slow Internet connection.

A number of companies today offer to periodically poll Web services using a geographically distributed set of clients. At best, polling can generate an *approximation* of the response time perceived by actual customers. At worst, the geographic and temporal distribution of the polls may be completely different from that of a web site's actual customers, leading to useless response time data. Ultimately, the only reliable way to obtain client-perceived response time is to actually measure it. Examples of such polling services include ServerCheck[4], GoldTest [5], and eValid [6].

This paper presents a novel framework for measuring the *actual* response time perceived by customers as they access a Web service. It does not require a third party, statistical polling, or extra workload. It is applicable to any business, whether they run their services "in-house" or in a consolidated server, and whether the content they serve is statically or dynamically generated. Our framework also works with the existing technology restrictions and limitations. In particular, it does not require any changes to the Hyper Text Transfer Protocol [7], client browsers, or any existing technology. The framework leverages the scripting and event notification mechanisms in HTML 4.01 [8] and uses scripting languages such as JavaScript 1.1 [9] to measure and collect client-perceived response times without any browser modifications. Both HTML 4.0

and JavaScript 1.1 are fairly de facto standards and are supported by the popular Netscape Navigator and Internet Explorer browsers. Finally, our methods do not depend on the use of Java applets [10] and cookies [11], support for which could be disabled by users.

The response time data that we collect can be used in a variety of ways, some of which we describe here. A Web service may use the data to determine whether it is in danger of losing its customers due to intolerable response times. It may also use the data to decide whether a proxy caching service will be useful, and if so, where to place the proxy caches. Web hosting IDCs contract with businesses through Service Level Agreements (SLA), which specify the quality of service that the IDC will provide. The ability to collect accurate client-perceived response time values also enables the creation of a new class of SLAs in which an IDC can contract to serve some fraction of the site's visitors within a specified amount of time. Finally, accurate response time values can be used to differentiate between different proxy caching services.

We begin by providing some background in Section 2 and describe the framework implementation in Section 3. In Section 4, we analyze real response time data we have obtained by instrumenting "The Wondering Minstrels" web site. We describe related work in Section 5 and our conclusions in Section 6.

2. Background and Overview

We begin by defining some basic terms used through the following discussion. A *bundle* is a set of web pages that have been instrumented to measure client-perceived response times. These may be HTML or non-HTML pages, and could be static files or dynamically generated content. The HTML pages in the bundle may contain links to each other, enabling users to traverse the bundle by dereferencing hyperlinks. There can be two kinds of links: *instrumented* and *uninstrumented*. An *instrumented* link points to a page whose response time we want to measure when the link is dereferenced. A page pointed to by an instrumented link is itself instrumented. An *uninstrumented* link points to a page for which we do not want to know the response time.

Users arrive at a specific page within a bundle in one of two ways, causing it to be loaded in their browser. First, the user may have dereferenced an instrumented link from another page within the bundle. We call this an *instrumented* entry into the page. Alternatively, they may have directly entered the page's URL into the browser, or may have followed a link from a page outside the bundle. We term this an *outside* entry into the page.

The framework introduced here allows accurate measurements of all instrumented entries to HTML pages in a bundle. Furthermore, it calculates the response times of each embedded object such as images or Flash animations [12] within a web page.

2.1. HTML Scripts and Event Handlers

The HTML standard specifies that a HTML page can contain embedded client-side *scripts*, which are executed as the page is parsed by the browser[1]. Furthermore, HTML link elements can contain a script snippet instead of a URI. When a link containing a script snippet is dereferenced, the script is executed. The following paraphrases the HTML standard's script description [8].

> A client-side script is a program that may accompany an HTML document or be embedded directly in it. The program executes on the client's machine when the document loads, or at some other time such as when a link is activated. Authors may attach a script to a HTML document such that it gets executed every time a specific event occurs. Such scripts may be assigned to a number of elements via the intrinsic event attributes. Script support in HTML is independent of the scripting language.

Browsers today support a wide variety of scripting languages. JavaScript [9], VBScript [13], and Tcl [14] are three examples of popular scripting languages. JavaScript is by far the most popular scripting language, and is supported by virtually all browsers that allow scripting [15].

The HTML standard also specifies several intrinsic *events* and the interfaces through which a client-side scripts can be invoked when different events occur. The Timekeeper uses three specific events that can invoke a script attached to a document:

- onclick(): In the context of a link, the onclick handler is invoked when the user clicks on a link. The link is de-referenced based on the handler's return value.
- onload(): In the context of a document, the onload handler is triggered when a document and its embedded elements have been fully loaded. In the

[1] A browser can defer script parsing only if the "defer" attribute of the SCRIPT tag is set true. In it absence, scripts must be parsed as they are encountered.

context of an OBJECT element, the onload handler is triggered when the object is fully loaded.

- onunload(): Triggered when a document is about to be unloaded to make room for a new document, or when the browser is being closed.

2.1. Overview

Abstractly, our scheme consists of the following four basic steps. We describe the details of each step later in the paper.

1. Before sending a request to a web server, the client browser is made to determine and remember the current time. This action is performed using a client-side script embedded within the currently displayed page.
2. The normal browser actions cause the requested page and its embedded elements to be loaded and displayed.
3. After the response is fully received, the client browser computes the response time as the difference between the current time and the start time remembered from step 1. We again use a client-side script to carry out the computation, initiating this step through an onload event handler that triggers when the page fully loads. Note that since both the start and end times are computed on the client, no clock synchronization is required.
4. Once a response time value is determined, client-side script action transmits it to a "record keeper" web site based on an established policy. Several policies may be implemented. For instance, response time samples could be submitted when they are collected, submitted in bulk, or submitted only when they are above a particular threshold value. The record keeper could be any web server on the WWW, enabling an agent other than the origin web server to collect and maintain response time values.

Steps 1, 3, and 4 are divided between two agents within the browser called the *Timekeeper* and the *Librarian*:

- The *Timekeeper* is responsible for computing the response time values. In doing so, it uses the *Librarian* to save and retrieve state values. Depending on a prespecified policy, the Timekeeper also communicates the computed response time samples along with other information to a record keeping web site.
- The *Librarian* stores and retrieves state values upon request. It provides the Timekeeper with a well-defined interface for performing these actions.

The following sections discuss these mechanisms in detail. We use JavaScript as the scripting language in our prototype implementation. Originally introduced by Netscape, JavaScript has become the de facto scripting language on the WWW, supported by most popular browsers including Navigator and Explorer. However, our approach is equally valid with other scripting languages such as VBScript[13] or Tcl [14].

3. Implementation

3.1. Timekeeper

The Timekeeper determines the current time and computes time deltas. It occasionally uses the Librarian to check in new time samples and to check them out later. It carries out these actions using HTML events and scripts. The Timekeeper can also be made to record the identity of each instrumented page or image along with the corresponding response time sample.

Several policies may be implemented. For instance, response time samples could be submitted when they are collected, submitted in bulk, or submitted only when they are above a particular threshold value. Under some conditions, the record keeper can be any computer on the WWW. This could serve having a third party verify Service Level Agreements.

Consider an instrumented link in a web page. By definition, the link points to a page that is itself instrumented. When a user clicks on the link, a Timekeeper script is invoked. The script determines the current time, records it using the Librarian, and then permits the link to be dereferenced. After this page is fully loaded, it's onload handler is invoked. The handler obtains the time at which the page request was made by querying the Librarian. It then calculates the response time as the difference between the current time and the "send-time". If the measured response time is to be transmitted later, the handler records it using the Librarian. Figure 1 summarizes these actions.

When the user directly types the location of a document in the address bar of a browser, the currently displayed document in the browser window is replaced with the one requested by the user. Before loading the new document, the browser invokes an onunload handler, if one has been specified. The Timekeeper performs cleanup operations on this event, communicating the collected response time values to a record keeper. A cleanup operation is also initiated when the user closes the browser window.

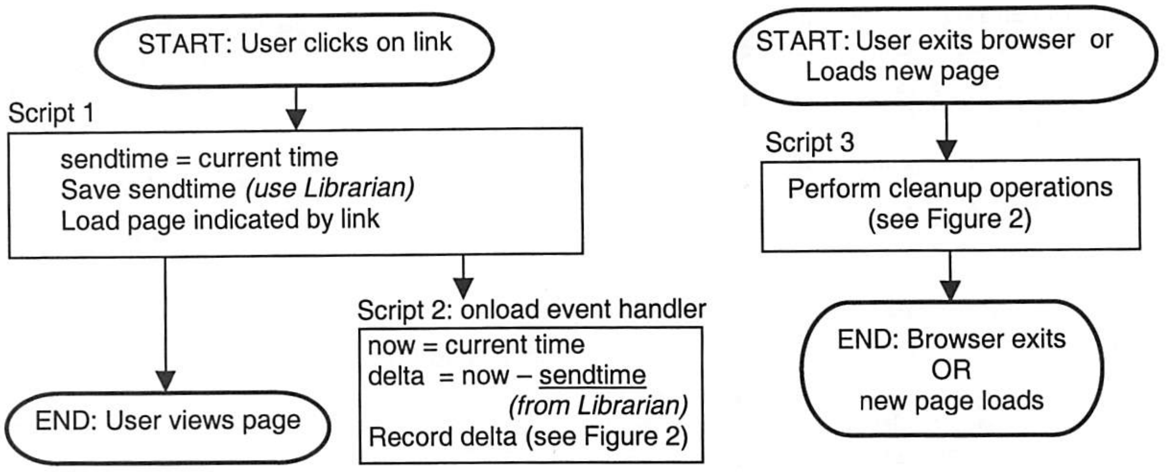

START: User clicks on link
Script 1
sendtime = current time
Save sendtime (use Librarian)
Load page indicated by link
END: User views page
Script 2: onload event handler
now = current time
delta = now – sendtime (from Librarian)
Record delta (see Figure 2)
START: User exits browser or Loads new page
Script 3
Perform cleanup operations (see Figure 2)
END: Browser exits OR new page loads

Figure 1: Timekeeper actions

The cleanup operations performed by the Timekeeper and its actions for informing the record keeper are summarized in Figure 2. There are two points of entry shown in the figure, corresponding to the unexplained actions in Figure 1. The left hand side depicts the Timekeeper actions on individual response time samples. The right hand side shows the Timekeeper actions if the browser window is closed or if the user directly loads a new page.

The Timekeeper may implement any desired policy to communicate the response time samples. For instance, in order to amortize the cost of communicating the response time samples to the record keeper, a policy to collect some number of samples before communicating them could be implemented. Alternately, the policy may be to communicate only response times greater than an upper bound. The Timekeeper actions in Figure 2 implement the former ("communicate samples in bulk") policy. The next section describes the Timekeeper actions to communicate the collected response time samples in greater detail.

3.2 Transmitting Response Time Values

The response time samples collected by the Timekeeper and saved by the Librarian must be communicated to a *record keeper* in order to be of use. Our framework permits the record keeper can be the source web server itself, or any other web server on the Internet. Several policies could be established for communicating the samples. Some examples are:

- Communicate on every sample
- Communicate after accumulating a prescribed number of samples

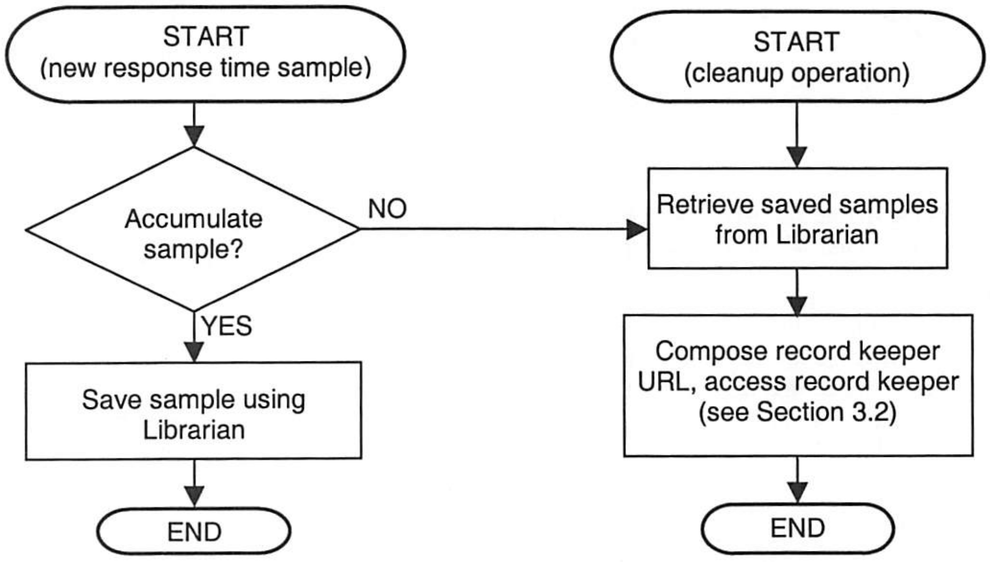

START (new response time sample)
Accumulate sample?
NO
YES
Save sample using Librarian
END
START (cleanup operation)
Retrieve saved samples from Librarian
Compose record keeper URL, access record keeper (see Section 3.2)
END

Figure 2: Timekeeper operations contd. (implements "communicate samples in bulk" policy)

- Communicate only those samples that exceed a pre-established threshold value.
- Communicate only samples from favored customers (as determined by a cookie).

For any established policy, we assume that the source web server provides the Timekeeper with a JavaScript function to determine when and what samples to communicate.

The Timekeeper uses a JavaScript Image object in order to perform the communication. Image objects were introduced in client—side JavaScript 1.1 and are primarily used to preload images [9]. By setting the src attribute of an image object to a desired URL, the specified content is loaded and placed in the browser's cache. By initiating actions such as image replacement and animation only after all the desired images are preloaded, the quality of visual effects in the browser can be enhanced. Image loading does not block JavaScript execution. Instead, the image content is loaded asynchronously alongside other browser actions.

When the Timekeeper needs to communicate with the record keeper, it composes a special URL containing the data. This URL accesses a script on the record keeper site. The record keeper extracts information from the URL, and replies with a response. The response is set to be non-cacheable in order to prevent intervening proxies from squelching record keeper communication.

HTML specifications stipulate that if the currently loaded document has an unload event handler specified, that handler must be invoked before a new document is loaded. The unload handler is also invoked if the user closes the browser window. The Timekeeper uses this event handler to communicate with the record keeper when the user visits an uninstrumented page or makes an outside entry to a page in the bundle.

Using the image preload technique to communicate with the record keeper on a browser close is problematic. Consider a case where the user has closed the browser, and the unload handler has initiated the record keeper image load. Since image loading is done asynchronously, the browser may die even before the TCP connection to the record keeper is fully open. Unfortunately, to the best of our knowledge, neither Navigator nor Explorer support UDP URL schemes. Consequently, we use a different method for record keeper communication on a browser close.

In both Navigator and Explorer, a new browser window can be reliably opened from an unload handler. A new window is opened by using the open method of the current window object[2]. We use this feature to communicate with the record keeper by opening a new window with the response time URL. The record keeper is set up to respond with a page that closes the window. This is done with a page with an onload handler that performs a "window.close( )" operation.

Since the unload handler is invoked when each document is displaced from the browser window, the Timekeeper needs to distinguish between the unload handler invocations on an instrumented entry to the next page, or on an outside entry to an arbitrary page that may be inside or outside the bundle. We make the distinction by setting a variable in the window object before carrying out an instrumented entry to the next page. The unload handler checks whether this variable is set in the window object. If it finds the variable set, we know that the current page is being displaced to make room for an instrumented page in the bundle. Otherwise, the Timekeeper performs the cleanup operations from within the unload handler. We reset this variable in readiness for the next unload within the onload handler that is invoked after an instrumented page has fully loaded.

Since opening a separate communication window could detract from the end-user experience, the Timekeeper has two choices. First, it can make the communication window small (but see Section 3.3.2). Second, it can decide that losing the last few collected samples is better than opening a new window.

Yet another option is to use a cookie. If the client browser has cookies enabled, the Timekeeper could also save the information in a cookie for transmission at a later date. This cookie could be expired as soon as a request carrying it is sent to the source web server, causing it to no longer be sent out on future requests. Due to JavaScript security restrictions that control access to cookies from scripts, the record keeper will have to be the same as the source web server in this case.

3.3. The Librarian

The Librarian is responsible for storing and retrieving time samples on behalf of the Timekeeper. At present, to the best of our knowledge, there is no straightforward mechanism in a browser to maintain state across page loads. In particular, for security reasons, browsers do not permit client-side scripts to maintain state across

[2] For example, enable JavaScript in your browser and visit http://www.cs.rice.edu/~rrk/neverclose.html. Many web sites use this annoying technique to make it difficult for users to leave their site.

```
function OpenStateWindow( )
{ // Open a state window if needed
    var h = self.open ("", "statewin",
            "width=100,height=100,location=no");
    if (typeof h.valid == "undefined") {
        h.document.write ("Benign msg for user");
        h.document.close ( );
        h.valid = true;  // Don't write next time
    }
    return h;
}

function SaveSendtime ( ) {
    var h = OpenStateWindow ( );
    h.sendtime = (new Date( )).getTime ( );
}

function GetSendtime ( ) {
    var h = OpenStateWindow ( );
    return h.sendtime;    // can be undefined
}

function RTonload ( ) { // onload handler
    var sendtime = GetSendtime ( );
    if (typeof sendtime != "undefined") {
        var now = new Date ( );
        var delta = now.getTime ( ) – sendtime;
        // Accumulate delta, or transmit now
    }
}

function RTonclick ( ) { // link onclick handler
    SaveSendtime ( );
    return true; // Permit link to be dereferenced
}
```

Figure 3: Timekeeper operation with Separate Window Librarian

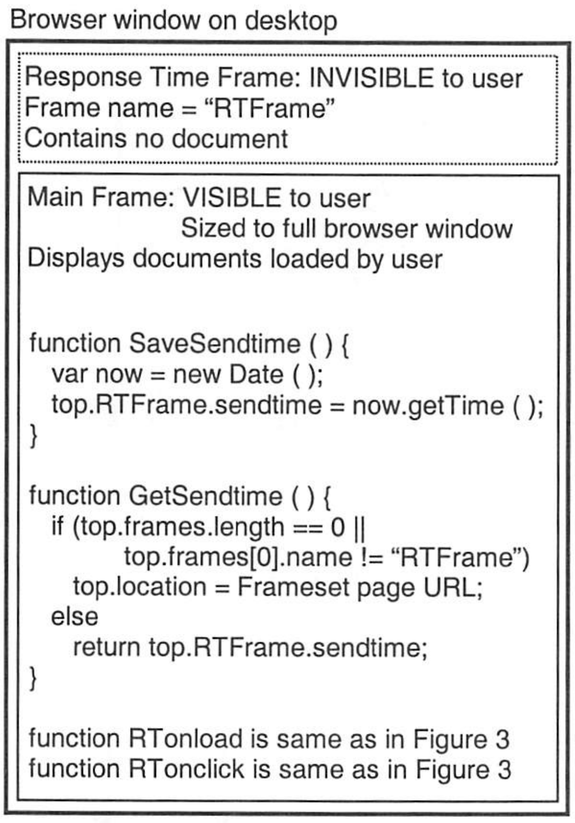
page loads. When a new document is loaded, all of the scripts and variables associated with a page are cleared. In this section, we present three approaches for implementing the Librarian without resorting to browser modifications.

3.3.1 Saving State in a Cookie

HTTP, the protocol used for retrieving web pages, is inherently stateless [7]. Cookies were introduced as a mechanism to enable clients to build stateful sessions on top of HTTP [11]. Simply stated, a cookie is a tag created by the server and delivered to the client along with an HTTP response. On subsequent requests to the same server, the client presents the tag along with its request. Cookies permit a session to be built using individual HTTP transactions.

Cookies can be subverted for maintaining state. While it is the server that typically sets cookies, client-side scripts also have the ability to set, modify, and retrieve cookies. Thus, the Librarian could use cookies to maintain state across page loads. The Librarian could use JavaScript when state needs to be saved, and retrieve it again using JavaScript.

The naïve approach described above has a severe drawback. Today, the WWW is dotted with caches and proxies. The idea behind caching is to place an object "closer" to the user, reducing the demands placed by a request on both the network and the origin web server, enabling it to be serviced quickly. However, since cookies are used to create sessions out of HTTP, a cache that observes a request with an attached cookie typically does *not* service the request forwarding it instead to the origin web server [7]. This is the correct approach since two requests for the same URL with different cookies could potentially lead to different responses. Consequently, using cookies to maintain state would cause each request to be sent to the origin web server, negating the usefulness of proxies and caches. The cookie approach is therefore practical only when the content being delivered is itself dynamic, with no intervening proxy caching it.


Browser window on desktop

Response Time Frame: INVISIBLE to user
Frame name = "RTFrame"
Contains no document

Main Frame: VISIBLE to user
Sized to full browser window
Displays documents loaded by user

```
function SaveSendtime ( ) {
  var now = new Date ( );
  top.RTFrame.sendtime = now.getTime ( );
}

function GetSendtime ( ) {
  if (top.frames.length == 0 ||
        top.frames[0].name != "RTFrame")
    top.location = Frameset page URL;
  else
    return top.RTFrame.sendtime;
}

function RTonload is same as in Figure 3
function RTonclick is same as in Figure 3
```

Figure 4: Timekeeper and Librarian operations when using frames

3.3.2 Using a Separate Window

Since client-side JavaScript enables state to be stored in a window object, the simplest approach is to open a new window on the first outside entry to a web page in the bundle and to save the state there. For as long as this window stays open and until a different URL is loaded in it, state stored in its context can be recovered. Figure 3 shows sample JavaScript code that achieves this goal.

One limitation of this approach is the presence of the state window on the user's desktop. Even though this window is small and can be made to contain a benign message, a user may arbitrarily close the window. Furthermore, JavaScript security restrictions prevent a script from opening a window in a minimized state, or with a size smaller than a prescribed minimum, unless the script has the UniversalBrowserWrite privilege. While this privilege can be obtained by involving the user, the process is fairly awkward and cumbersome.

3.3.3 The Frame Approach

In the previous section we described how a separate browser window on the user's desktop could be used to save state. However, in client-side JavaScript, the *window object* does not have a one-to-one correlation with a browser window. More specifically, each HTML *frame* within a browser window corresponds to a separate window object. This correspondence paves the way for us to save state in the window object context of a frame *within* the same browser window as that displaying the document being loaded by the user.

The window object is the global object and execution context in client-side JavaScript. There is no direct correlation between a *window* as viewed in a desktop environment, and the *window object*. A window object is created for each desktop-level window *or* frame within a browser desktop window that displays a HTML document. From our perspective, the interesting property of client-side JavaScript is that it permits scripts executing in the context of one window object to access variables and scripts executing in another window object's context [9].

The HTML standard describes frames as follows, permitting frames to be created with zero size [8]. Such frames will be invisible to the user.

> HTML frames allow authors to present documents in multiple views, which may be independent windows or subwindows. Multiple views offer designers a way to keep certain information visible, while other views are scrolled or replaced. For example, within the same window, one frame might display a static banner, a second a navigation menu, and a third the main document that can be scrolled through or replaced by navigating in the second frame.

The Librarian can use frames to save state by placing each instrumented page in the bundle within a frameset document that divides the top-level browser window into two frames: the response-time frame and the main frame. The response-time frame is set to zero size and is therefore not visible to the user. The main frame holds the instrumented content.

There are two ways a user can make an uninstrumented entry to a page. First, a user could directly enter the URL of the frameset document. This causes both the response-time frame and the data frame to be loaded within the browser. Alternately, the user could directly enter the URL of the data frame in the browser's location bar. To force the browser to load the corresponding frameset document, each instrumented page in the bundle contains JavaScript to check for the existence of the response-time frame. The existence check is made after the main frame has loaded. If the response time frame does not exist, the JavaScript forces the corresponding frameset document to be loaded in the top-level browser window.

State saving is accomplished exactly as in the case with the separate window. Figure 4 explains the approach, showing how the Timekeeper and the Librarian interact together to determine the response time values.

Visiting the site through a browser window causes the following actions to take place. On the first outside entry to an uninstrumented page, the JavaScript actions in Figure 4 cause the corresponding frameset document to be loaded. As long as the user makes instrumented entries to the other pages in the bundle, the response time frame stays in the top-level browser window. The response-time frame needs to be reloaded only if the user makes an uninstrumented entry to a page in the bundle.

The main limitation of this approach is the need for loading the frameset document on uninstrumented entries to pages in the bundle. When encountering a page with frames, the browser first obtains the "container" frameset document. Only after receiving the container frameset can the browser determine what documents to obtain and render in the internal frames. This could give rise to extra client-server transactions and delays for the user.

Three factors mitigate the problem caused by the frameset document. First, the web site might already have a frame that exists on all pages, enabling the Librarian to simply use that frame. Second, the extra client-server transactions (when the Librarian uses a dedicated frame) are encountered only during the *initial* uninstrumented entry to a web page. Subsequent instrumented browsing occurs at full speed. Finally, it may be possible to avoid the extra transactions by setting a long lifetime for the frameset documents. HTTP permits content to be delivered with explicit expiration times, allowing intervening caches and the client to cache content and use it *without checking for validity* against the origin web server. By providing the frameset document with a long lifetime, the browser needs to fetch the container frameset only when it is absent from the local browser cache. Consequently, when the user revisits a bundle at periodic intervals, as often happens since people are creatures of habit, the browser will be able to use a cached copy of the frameset document. Only the main frame's content will need to be fetched in such cases.

3.3.4 Using the Window Name

In Section 3.3.3 we described the window object in client-side JavaScript. Every window object has a name property. This property exists primarily for use as the value of a HTML TARGET attribute in the <A> or <FORM> tags. In essence, the TARGET attribute enables an anchor or form to display its results (when the linked document is dereferenced or the form is submitted) in the window with the supplied name.

The initial window and all new browser windows opened by most versions of both Explorer and Navigator have no pre-defined name property. Consequently, these windows cannot be addressed with a TARGET attribute. The name property is read-only in JavaScript 1.0, creating a problem when the initial window has to be addressed. JavaScript 1.1 resolves this problem by enabling the name attribute to be modified from within a script [9].

When a new page is loaded in a window, all of the scripts and variables associated with the window object are cleared. However, a window's name property *persists* across page loads. Thus a web page loaded into a window can be the target of actions in another window, even if the loaded content does not explicitly set the window name. This feature enables the same content to be loaded in the main browser window or in a popup window, depending on the context from where it is referenced.

We can leverage the persistence of the name property to store state. The idea is to append the desired state to the window name and to retrieve it from there, restoring the name after the retrieval[3]. Figure 5 illustrates the details of this approach.

```
function SaveSendtime ( ) {
   var s = (new Date( )).getTime ( );
   window.name += '_RT_' + s;
}

function GetSendtime ( ) {
   var n = window.name;
   var rt = n.indexOf ('_RT_');
   var s = n.substring (rt+4, n.length);
   window.name = n.substring (0, rt);
   return s;
}

function RTonload is same as in Figure 3
function RTonclick is same as in Figure 3
```

Figure 5: Timekeeper and Librarian operations when using window names

An obvious limitation of using the window name is the race condition it introduces. In order to compute response times, the window name is changed just prior to a new document being loaded in a window. The saved state is retrieved and the window name restored only when the document has fully loaded. During this interval, the window cannot be referred to by its original name.

The race condition causes a problem partly due to the awkward interface that client-side JavaScript provides for referring to a window. Given a window name, the only way to obtain a reference to the window is by using the window.open method. The name supplied to the method must be exact, with no wildcards allowed. Furthermore, if a window with the supplied name does not exist, the method simply opens a new window with that name. These properties of the open interface imply that if a window with an ongoing instrumented page load is targeted, a new window with the supplied name is opened. This may distract and confuse the user.

In the next section, we describe how the window name approach can be combined with the use of separate windows to yield a practical, useful solution.

3.3.5. Separate Windows + Window Names

Taken individually, the separate window and window name schemes suffer from limitations when used to implement the Librarian. The separate window could be a distraction for the user, since it must remain open for

[3] We have empirically determined that both Navigator and Explorer support names over 1000 characters.

as long as there are instrumented entries to the bundle. The window name scheme exposes a race condition that could open a new browser window on the user's desktop, contrary to the content developer's intent.

However, the separate window and the window name schemes can be combined together to yield a practical solution that does not suffer from these limitations. The idea is to divide the windows in which the instrumented pages will be displayed into two sets: the *main* windows and the *child* windows. Main windows are those that are not targets for any content. Child windows are those in which content is loaded by actions in both main and child windows.

For a page that is loaded in a main window, the Librarian saves state in that window's name. For a page that is loaded in a child window, the Librarian saves state as a property in the context of the child window's ancestor. The ancestor can be determined by following the opener property of the child window, which is a reference to the window object that opened the child window. State saving and retrieving will be accomplished by using a combination of the scripts shown in Figures 4 and 5. Note that it is totally safe to change a main window's name, since by definition, a main window can never be the target for any content.

3.4. Limitations

Our schemes have the following limitations:

1. We cannot compute response times for outside entries to instrumented web pages. For example, we cannot compute response times for pages loaded by directly entering a URL into a browser's location bar.
2. Our scheme handles instrumented entries to HTML pages and objects embedded within HTML pages. We cannot compute response times for other MIME types. For instance, we cannot compute the response times for pure images (outside of a HTML document) directly loaded by a browser, or for PDF documents loaded and displayed by the Adobe plugin. In order to handle such MIME types, we need plugin support in the form of a public method that can be used to determine the load status of the content. Flash animations are an example of a MIME type we can handle even when the animation is loaded outside of an <OBJECT> context, by using the PercentLoaded public method of the MacroMedia Flash plugin [16].
3. Our implementation uses client-side JavaScript 1.1. Consequently, it will not work with browsers that do not support JavaScript 1.1, or that have disabled JavaScript. In particular, our implementation requires Navigator versions above 3.x, and Explorer versions above 4.0.

4. Status and Evaluation

We have implemented our scheme on "The Wondering Minstrels," a poem-of-the-day web site that receives several hundred hits a day. The site can be accessed at http://www.cs.rice.edu/~rrk/minstrels.html, and has over 650 poems (now) that range in size from 2.6KBytes to 30 Kbytes. A few files that index all the poems take up about 230Kbytes. We instrumented this site using an automated tool we have developed, setting the record keeper to be www.cs.utexas.edu. Note that the origin web server and the record keeper are different servers.

	Any time	00:00 to 07:59	08:00 to 15:59	16:00 to 23:59
All	971	161	411	399
Unknown TLD	92	25	44	23
TLD: .us, .com, .org, .net, .edu, .ca	685	72	285	328
All other TLDs	194	64	82	48

Table 1: Distribution of collected response time values

Table 1 shows the distribution of the response time values we have collected from real visitors to the site for files between 4400 and 4900 bytes in size. As of Jan 21, 2001, these were the most frequently accessed files from the site. The actual web server response is larger by about 260 bytes, which is the approximate size of the HTTP header for these responses. We have broken down the collected values by the time when the request was serviced, and by the top-level domain (TLD) of the client making the request. All times listed are behind UTC by 6 hours. We were unable to resolve hostnames for a number of clients accessing the site. These unresolved hostnames are listed as "Unknown" and account for 9.5% of the collected values.

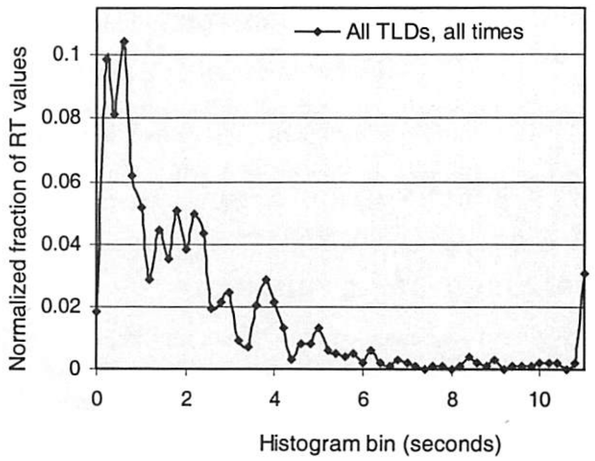


Figure 6: Distribution of collected response time values

Figure 6 shows a histogram of the normalized response time values, irrespective of the TLD from where or the time the request was made. Each data point in the figure indicates the number of response time values that fall into a 200 ms bin to the right of that point. All response times exceeding 10.8 seconds are aggregated and placed into the bin at 11 seconds. There are several interesting points to note about the figure. First, a small fraction of the requests (about 1.8%) were satisfied within 200ms. This is probably due to a cache hit in the client itself, or in a proxy very close to it. Second, by considering the average attention span window to be 4 seconds (i.e., "I want my page to load in 4 seconds or I get bored"), we see that 16% of the responses were served outside the average attention span window.

Breaking down the response time values by time period illustrates the variation in response times over the course of a day. Figures 7, 8, and 9 show the response time values for the midnight–8AM, 8AM–4PM, and 4PM-Midnight time windows respectively. These figures show that the fraction of pages that take longer than 4 seconds to load decreases from 22% in the night, to 19% during the morning, and to 11% during the afternoon.

Figures 10 and 11 break up the response time values by top-level domain. We group together all com, org, edu, net, us, and ca domains into one group, and all other known TLDs into another. We realize that com, org, and net clients could be spread throughout the world. However, this crude division allows us to focus on the response times faced by visitors located far away from the Minstrels web site.

Figure 11 clearly shows the larger response times seen by far-away visitors. About 26% of the users in this group faced delays of more than 4 seconds. Further analysis reveals that the two largest set of visitors in this

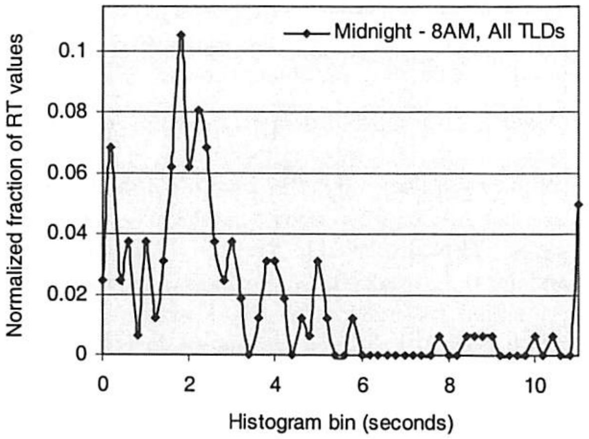


Figure 7: Response time distribution from 0000-0759 CST

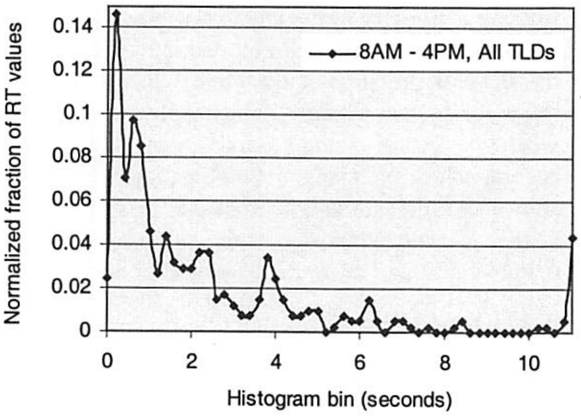


Figure 8: Response time distribution from 0800-1559 CST

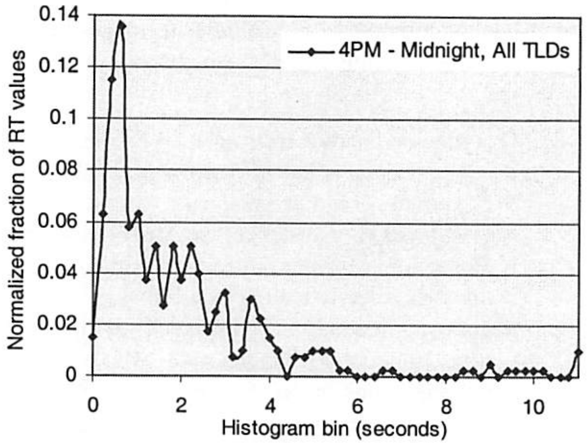


Figure 9: Response time distribution from 1600-2359 CST

group are from the United Kingdom and Argentina. While only 12.5% of the English faced response times larger than 4 seconds, fully 58% of the Argentines fell into the same category. Interestingly, if the Argentine visitors were all served within 4 seconds, only 15% of the far-away visitors would have waited for more than 4 seconds. These results indicate that if the Minstrels site were to contract with a proxy caching service, placing just one cache in close proximity to the Argentine visitors would provide the most benefit.

The nature of analysis presented here is similar to what a commercial site would want to conduct on its visitors. Our framework enables such an analysis and shows how a site appears to different user groups accessing it. We would like to stress here that such an analysis would not be possible using information gathered at the server only, as it would lack the actual networking effects. Also, we would like to stress that traditional approaches such as pinging the server would have a prohibitive cost emulating all possible users and their geographic distribution. Our experience with measuring actual response times seen by real visitors to the Minstrels site illustrates the need for a measurement framework such as ours. The analysis we describe here is one of many that can be carried out with real response-time data.

5. Related Work

A number of web sites contract with third party companies to poll their servers at periodic intervals. Typically, a battery of geographically distributed clients "ping" the server with fictitious transactions. The site owner specifies the frequency of polling and the geographic distribution. Examples of such polling services include ServerCheck™ [4], GoldTest™ [5], and eValid™ [6]. Polling suffers from several drawbacks. First, the data obtained through polling is a statistical approximation to the response time seen by real visitors to the web site. Polling can also increase the load generated on a site's servers. Ensuring accurate or complete geographic coverage using polling is also difficult. Finally, some services such as financial transactions may be cumbersome to measure using fictitious requests.

Candle's eBusiness Assurance (eBA™) product [17] provides an accurate breakdown of the time spent by a user on a web page. eBA uses an applet to store state on the client browser between page loads. The time at which the user clicks on a link is stored in the applet's context and the response time computed after the page fully loads within the browser. eBA appears to have been introduced around September 2000. In comparison to our work, eBA's main difference is its dependence on a Java applet in order to save state. Since applets are still not widely used owing to their (as perceived) heavyweight nature, this is a fairly significant drawback. In contrast, our scheme requires only JavaScript support in order to compute response times. A significant fraction of the sites on the web today (IBM, CNN, ETrade, CNBC, Disney, etc.) use JavaScript, greatly reducing the barrier to using a scheme such as ours.

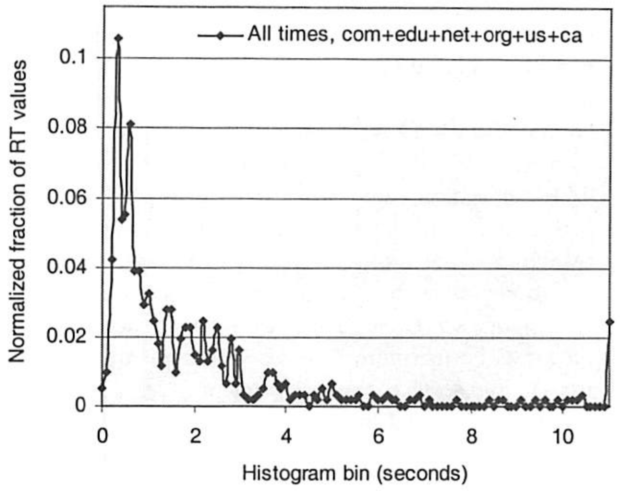


Figure 10: Response time distribution for shown TLDs

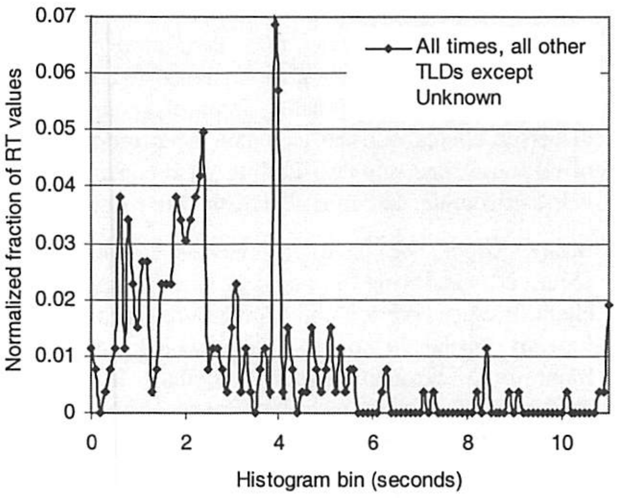


Figure 11: Response time distribution of "far away" TLDs

Tivoli has also introduced a product around September 2000 called Quality of Service Monitor (QoS), as part of their Web Management Solutions package [18]. QoS uses a proxy that sits inside the web site firewall to intercept responses as they leave the site. The proxy adds a small amount of JavaScript code to the content and also remembers the time at which the response is sent back. The added script contacts the QoS proxy when the page has finished rendering. QoS then approximates the response time seen by the client as the delta between when the response is sent to the client, and when the proxy hears back from the client. There are two main differences between the QoS work and ours. First, in order to avoid clock synchronization problems, the record keeper must be the QoS proxy. Second, QoS calculates the response time from the point of view of the proxy, giving rise to three important implications. First, response times can be computed only if the client contacts the proxy after each page is rendered, adding an extra client--proxy communication to every transaction. Second, the client most likely does not have to create a new TCP connection to communicate with the proxy after rendering the page. Consequently, the measured response time does not take TCP connection setup times into account. Third, QoS cannot determine response times for requests fulfilled by intermediate proxy caches. QoS is more suited for environments such as e-commerce transactions, where the origin server is likely to be involved in every response. Furthermore, QoS also measures the back-end transaction service time (by tagging requests as the enter the site), and the page render time (measured by the added script), both of which are very valuable.

6. Conclusions

On the World Wide Web, the response time seen by a client is a key metric that determines end-user satisfaction. Most schemes in existence today rely on polling the web server using a set of geographically dispersed clients in order to obtain a representative set of response time samples. Polling yields data that is at worst inaccurate, and at best, statistical in nature.

In this paper, we have presented a framework for accurately measuring the response time perceived by a client browser. We are able to measure response times for all visits to pages instrumented using our framework, through hyperlinks that have been instrumented. We can also measure response times for all objects embedded within a web page. Our framework imposes very little overhead on the client computer, and fits the needs of various *existing* commercial web sites.

We divide the work in collecting response time samples between a *Timekeeper*, who computes all time values, and a *Librarian*, who provides the Timekeeper with services for saving and retrieving state. The Timekeeper implementation is fairly straightforward. On the other hand, implementing the Librarian is challenging owing to the difficulty in persisting state across page loads in existing browsers. We present several solutions to this problem, which are well suited for use in existing commercial web sites.

Acknowledgements

We would like to thank Sitaram Iyer for allowing us to instrument the Wondering Minstrels web site and allowing us to publish the aggregate data. We would also like to thank the Computer Science departments at Rice University and UT Austin for use of their web servers. Brian Pinkerton, our shepherd, made many valuable comments that have improved the quality of this paper. Finally, we would like to thank our anonymous reviewers for their helpful insights.

References

1. Akamai FreeFlow^{TM}, http://www.akamai.com/html/en/sv/content_delivery.html, Akamai Technologies Inc.
2. Digital Island Inc., Footprint^{TM}, http://www.digitalisland.net/services/cd/footprint.shtml
3. Inktomi Corporation, www.inktomi.com
4. NetMechanics ServerCheck^{TM}: http://www.netmechanics.com/monitor.htm, NetMechanics Inc.
5. GoldTest^{TM}: http://www.keynote.com/services/services/keyreadiness/goldtest.html, Keynote Systems Inc.
6. eValid^{TM} Test Services: http://www.soft.com/eValid/Services/Monitoring/index.html, Software Research Inc.
7. J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol – HTTP 1.1", RFC 2616, June 1999.
8. Dave Raggett, Arnaud Le Hors, and Ian Jacobs, "HTML 4.01 Specification", W3C, December 1999.
9. David Flanagan. JavaScript, The Definitive Guide, O'Reilly & Associates Inc., 1998
10. K. Arnold, J. Gosling, and D. Holmes, "The Java Programming Language", Addison-Wesley, 2000.
11. D. Kristol and L Montulli, "HTTP State Management Mechanism", RFC 2109, February 1997.
12. Macromedia Flash 5, http://www.macromedia.com/software/flash/, Macromedia Inc.
13. Microsoft Scripting Technologies: VBScript, http://msdn.microsoft.com/scripting/default.htm?/scripting/vbscript/default.htm, Microsoft Corporation.
14. John K. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley Publishing Company, 1994.
15. Microsoft Corporation, "Using VBScript and Jscript on a web page", http://msdn.microsoft.com/library/techart/msdn_vbnjscrpt.htm
16. Flash methods, http://www.macromedia.com/support/flash/publishexport/scriptingwithflash/scriptingwithflash_03.html, Macromedia Inc.
17. eBusiness Assurance^{TM}, http://www.candle.com/solutions_t/enduser_solutions/site_performance_analysis_external/index.html, Candle Corporation.
18. Tivoli Web Management Solutions, http://www.tivoli.com/products/demos/twsm.html, IBM Corporation

Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services

Kai Shen, Tao Yang, Lingkun Chu, JoAnne L. Holliday, Douglas A. Kuschner, and Huican Zhu
Department of Computer Science, University of California at Santa Barbara, CA 93106
{kshen, tyang, lkchu, joanne46, kuschner, hczhu}@cs.ucsb.edu

Abstract

Previous research has addressed the scalability and availability issues associated with the construction of cluster-based network services. This paper studies the clustering of replicated services when the persistent service data is frequently updated. To this end we propose Neptune, an infrastructural middleware that provides a flexible interface to aggregate and replicate existing service modules. Neptune accommodates a variety of underlying storage mechanisms, maintains dynamic and location-transparent service mapping to isolate faulty modules and enforce replica consistency. Furthermore, it allows efficient use of a multi-level replica consistency model with staleness control at its highest level. This paper describes Neptune's overall architecture, data replication support, and the results of our performance evaluation.

1 Introduction

High availability, incremental scalability, and manageability are some of the key challenges faced by designers of Internet-scale network services and using a cluster of commodity machines is cost-effective for addressing these issues [4, 7, 17, 23]. Previous work has recognized the importance of providing software infrastructures for cluster-based network services. For example, the TACC and MultiSpace projects have addressed load balancing, failover support, and component reusability and extensibility for cluster-based services [7, 12]. These systems do not provide explicit support for managing frequently updated persistent service data and mainly leave the responsibility of storage replication to the service layer. Recently the DDS project has addressed replication of persistent data using a layered approach and it is focused on a class of distributed data structures [11]. In comparison, we design and build an infrastructural middleware, called *Neptune*, with the goal of clustering and replicating existing stand-alone service modules that use various storage management mechanisms.

Replication of persistent data is crucial to achieving high availability. Previous work has shown that synchronous replication based on *eager* update propagations does not deliver scalable solutions [3, 9]. Various asynchronous models have been proposed for wide-area or wireless distributed systems [2, 3, 9, 15, 22]. These results are in certain parts applicable for cluster-based Internet services; however they are designed under the constraint of high communication overhead in wide-area or wireless networks. Additional studies are needed to address high scalability and runtime failover support required by cluster-based Internet services [6, 18].

The work described in this paper is built upon a large body of previous research in network service clustering, fault-tolerance, and data replication. The goal of this project is to propose a simple, flexible yet efficient model in aggregating and replicating network service modules with frequently updated persistent data. The model has to be simple enough to shield application programmers from the complexities of data replication, service discovery, load balancing, failure detection and recovery. It also needs to have the flexibility to accommodate a variety of data management mechanisms that network services typically rely on. Under the above consideration, our system is designed to support multiple levels of replica consistency with varying performance tradeoffs. In particular, we have developed a consistency scheme with staleness control.

Generally speaking, providing standard system components to achieve scalability and availability tends to decrease the flexibility of service construction. Neptune demonstrates that it is possible to achieve these goals by targeting partitionable network services and by providing multiple levels of replica consistency.

The rest of this paper is organized as follows. Section 2 presents Neptune's overall system architecture and the assumptions that our design is based upon. Section 3 describes Neptune's multi-level replica consistency scheme and the failure recovery model. Section 4 illustrates a prototype system implementation and three service deployments on a Linux cluster. Section 5 evaluates Neptune's performance and failure management using those three services. Section 6 describes related work and Section 7 concludes the paper.

2 System Architecture and Assumptions

Neptune's design takes advantage of the following characteristics existing in many Internet services: 1) **Information independence**. Network services tend to host a large amount of information addressing different and independent categories. For example, an auction site hosts different categories of items. Every bid only accesses data concerning a single item, thus providing an intuitive way to partition the data. 2) **User independence**. Information accessed by different users tends to be independent. Therefore, data may also be partitioned according to user accounts. Email service and Web page hosting are two examples of this. With these characteristics in mind, Neptune is targeted at *partitionable* network services in the sense that data manipulated by such a service can be divided into a large number of independent data partitions and each service access can be delivered independently on a single partition; or each access is an aggregate of a set of sub-accesses each of which can be completed independently on a single partition. With fast growing and wide-spread usage of Internet applications, partitionable network services are increasingly common.

Neptune encapsulates an application-level network service through a service access interface which contains several RPC-like access methods. Each service access through one of these methods can be fulfilled exclusively on one data partition. Neptune employs a flat architecture in constructing the service infrastructure and a node is distinguished as a client or a server only in the context of each specific service invocation. In other words, a server node in one service invocation may act as a client node in another service invocation. Within a Neptune cluster, all the nodes are loosely connected through a well-known publish/subscribe channel. Published information is kept as soft state in the channel such that it has to be refreshed frequently to stay alive [16]. This channel can be implemented using IP multicast or through a highly available well-known central directory. Each cluster node can elect to provide services through repeatedly publishing the service type, the data partitions it hosts, and the access interface. Each node can also choose to host a Neptune client module which subscribes to the well-known channel and maintains a service/partition mapping table. Services can be acquired by any node in the cluster through the local Neptune client module by using the published service access interface. The aggregate service could be exported to external clients through protocol gateways.

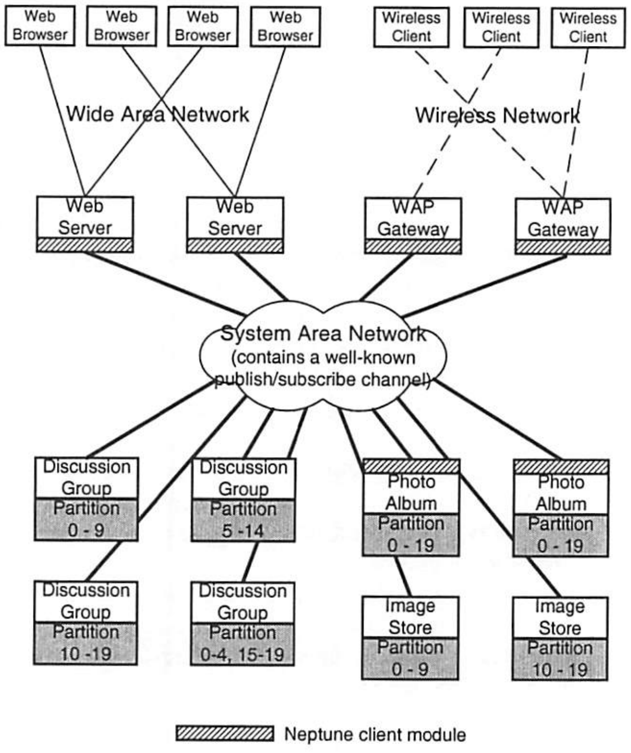

Web Browser
Web Browser
Web Browser
Web Browser
Wireless Client
Wireless Client
Wireless Client
Wide Area Network
Wireless Network
Web Server
Web Server
WAP Gateway
WAP Gateway
System Area Network (contains a well-known publish/subscribe channel)
Discussion Group Partition 0 - 9
Discussion Group Partition 5 - 14
Photo Album Partition 0 - 19
Photo Album Partition 0 - 19
Discussion Group Partition 10 - 19
Discussion Group Partition 0-4, 15-19
Image Store Partition 0 - 9
Image Store Partition 10 - 19
Neptune client module

Figure 1: Architecture of a sample Neptune service cluster.

Figure 1 illustrates the architecture of a sample Neptune service cluster. In this example, the service cluster delivers a discussion group and a photo album service to wide-area browsers and wireless clients through web servers and WAP gateways. All the persistent data are divided into twenty partitions according to user accounts. The discussion group service is delivered independently while the photo album service relies on an internal image store service. Therefore, each photo album node needs to host a Neptune client module to locate and access the image store in the service cluster. A Neptune client module is also present in each gateway node in order to export internal services to external clients. The loosely-connected and flat architecture allows Neptune service infrastructure to operate smoothly in the presence of transient failures and through service evolution.

2.1 Neptune Modules and Interfaces

Neptune supports two communication schemes between clients and servers: a request/response scheme and a stream-based scheme. In the request/response scheme, the client and the server communicate with each other through a request message and a response message. For the stream-based scheme, Neptune sets up a bidirectional stream between the client and the server as a result of the service invocation. Stream-based communication can be used for asynchronous service invocation and it also allows multiple rounds of interaction between the client and the server. Currently Neptune only supports stream-based communication for read-only service accesses because of the complication in replicating and logging streams. The rest of this section focuses on the support for the request/response scheme.

For the simplicity of the following discussion, we classify a service access as a *read access* (or *read* in short) if it does not change the persistent service data, or as a *write access* (or *write* in short) otherwise.

Neptune provides a client-side module and a server-side module to facilitate location-transparent service invocations. Figure 2 illustrates the interaction among service modules and Neptune modules during a service invocation. Basically, each service access request is made by the client with a service name, a data partition ID, a service method name, and a read/write access mode, as discussed below on the client interface. Then the Neptune client module transparently selects a service node based on the service/partition availability, access mode, consistency requirement, and runtime workload. Currently Neptune's load balancing decision is made based on the number of active service invocations at each service node. Previous work has also shown that locality-aware request distribution could yield better caching performance [14]. We plan to incorporate locality-based load-balancing schemes in the future. Upon receiving the request, the Neptune server module in the chosen node spawns a service instance to serve the request and return the response message when it completes. The service instance could be compiled into a dynamically linked library and linked into Neptune process space during runtime. Alternatively, it could run as a separate process, which would provide better fault isolation and resource control at the cost of degraded performance. Finally, since a data partition may be replicated across multiple nodes, the Neptune server module propagates writes to other replicas to maintain replica consistency.

We discuss below the interfaces between Neptune and service modules at both the client and the server sides:

- At the client side, Neptune provides a unified interface to service clients for seeking location-transparent request/response service access. It is shown below in a language-neutral format:
  $NeptuneRequest(NeptuneHandle, ServiceName,$
  $PartitionID, ServiceMethod, AccessMode,$
  $RequestMsg, ResponseMsg);$
  A $NeptuneHandle$ should be used in every service request that a client invokes. It maintains the information related to each client session and we will discuss it further in Section 3.1. The meanings of other parameters are straightforward.
- At the server side, all the service method implementations need to be registered at the service deployment phase. This allows the Neptune server module to invoke the corresponding service instance when a service request is received. In addition, each service has to provide a CHECK callback which allows the Neptune server module to check if a previously spawned service instance has been successfully completed. The CHECK callback is very similar to the REDO and UNDO callbacks that resource managers provide in the transaction processing environment [10]. It is only invoked during the node recovery phase and we will further discuss its usage and a potential implementation in Section 3.2.

2.2 Assumptions

We assume all hardware and software system modules follow the fail-stop failure model and network partitions do not occur inside the service cluster. Nevertheless, we do not preclude catastrophic failures in our model. In other words, persistent data can survive through a failure that involves a large number of modules or even all nodes. In this case, the replica consistency will be maintained after the recovery. This is important because software failures are often not independent. For instance, a replica failure triggered by high workload results in even higher workload in remaining replicas and may cause cascading failures of all replicas.

Neptune supports atomic execution of data operations through failures only if each underlying service module can ensure atomicity in a stand-alone configuration. This assumption can be met when the persistent data is maintained in transactional databases or transactional file systems. To facilitate atomic execution, we assume

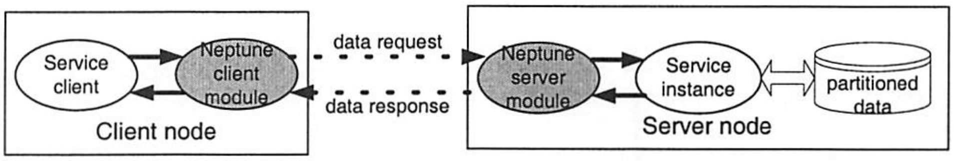


Figure 2: Interaction among service modules and Neptune modules during a request/response service invocation.

that each service module provides a CHECK callback so that the Neptune server module can check if a previously spawned service instance has been successfully completed.

3 Replica Consistency and Failure Recovery

In general, data replication is achieved through either *eager* or *lazy* write propagations [5, 9]. Eager propagation keeps all replicas exactly synchronized by acquiring locks and updating data at all replicas in a globally coordinated manner. In comparison, lazy propagation allows lock acquisitions and data updates to be completed independently at each replica. Previous work shows that synchronous eager propagation leads to high deadlock rates when the number of replicas increases [9]. In order to ensure replica consistency while providing high scalability, the current version of Neptune adopts a primary copy approach to avoid distributed deadlocks and a lazy propagation of updates to the replicas where the updates are completed independently at each replica. In addition, Neptune addresses the load-balancing problems of most primary copy schemes through data partitioning.

Lazy propagation introduces the problems of out-of-order writes and accessing stale data versions. Neptune provides a three-level replica consistency model to address these problems and exploit their performance tradeoff. Our consistency model extends the previous work in lazy propagation with a focus on high scalability and runtime failover support. Particularly Neptune's highest consistency level provides a staleness control which contains not only the quantitative staleness bound but also a guarantee of progressive version delivery for each client's service accesses. We can efficiently achieve this staleness control by taking advantage of the low latency, high throughput system-area network and Neptune's service publishing mechanism.

The rest of this section discusses the multi-level consistency model and Neptune's support for failure recovery. It should be noted that the current version of Neptune does not have full-fledged transactional support largely because Neptune restricts each service access to a single data partition.

3.1 Multi-level Consistency Model

Neptune's first two levels of replica consistency are more or less generalized from the previous work [5, 17] and we provide an extension in the third level to address the data staleness problem from two different perspectives. Notice that a consistency level is specified for each service and thus Neptune allows co-existence of services with different consistency levels.

Level 1. Write-anywhere replication for commutative writes. In this level, each write is initiated at any replica and is propagated to other replicas asynchronously. When writes are commutative, eventually the client view will converge to a consistent state for each data partition. Some append-only discussion groups satisfy this commutativity requirement. Another example is a certain kind of email service [17], in which all writes are total-updates, so out-of-order writes could be resolved by discarding all but the newest.

Level 2. Primary-secondary replication for ordered writes. In this consistency level, writes for each data partition are totally ordered. A primary-copy node is assigned to each replicated data partition, and other replicas are considered as secondaries. All writes for a data partition are initiated at the primary, which asynchronously propagates them in a FIFO order to the secondaries. At each replica, writes for each partition are serialized to preserve the order. This results in a loss of write concurrency within each partition. However, the large number of independent data partitions usually yields significant concurrency across partition boundaries. In addition, the concurrency among reads is not affected by this scheme.

Level 3. Primary-secondary replication with staleness control. Level two consistency is intended to solve the out-of-order write problem resulting from lazy propagation. This additional level is designed to address the issue of accessing stale data versions. The primary-copy scheme is still used to order writes in this level. In addition, we assign a version number to each data partition and this number increments after each write. The staleness control provided by this consistency level contains two parts: 1) **Soft quantitative bound.** Each read is serviced at a replica that is at most x seconds stale compared to the primary version. The quantitative staleness between two data versions is defined by the elapsed time between the two corresponding writes accepted at the primary. Thus our scheme does not require a global synchronous clock. Currently Neptune only provides a soft quantitative staleness bound and it is described later in this section. 2) **Progressive version delivery.** From each client's point of view, the data versions used to service her read and write accesses should be monotonically non-decreasing. Both guarantees are important for services like large-scale on-line auction when strong consistency is hard to achieve.

We explain below our implementations for the two staleness control guarantees in level three consistency. The quantitative bound ensures that all reads are serviced at a replica at most x seconds stale compared to the primary version. In order to achieve this, each replica publishes its current version number as part of the service announcement message and the primary publishes its version number at x seconds ago in addition. With this information, Neptune client module can ensure that all reads are only directed to replicas within the specified quantitative staleness bound. Note that the "x seconds" is only a soft bound because the real guarantee also depends on the latency, frequency and intermittent losses of service announcements. However, these problems are insignificant in a low latency, reliable system area network.

The progressive version delivery guarantees that: 1) After a client writes to a data partition, she always sees the result of this write in her subsequent reads. 2) A user never reads a version that is older than another version she has seen before. In order to accomplish this, each service invocation returns a version number to the client side. For a read, this number stands for the data version used to fulfill this access. For a write, it stands for latest data version as a result of this write. Each client keeps this version number in a $NeptuneHandle$ and carries it in each service invocation. The Neptune client module can ensure that each client read access is directed to a replica with a published version number higher than any previously returned version number.

3.2 Failure Recovery

In this section, we focus on the failure detection and recovery for the primary-copy scheme that is used in level two/three consistency schemes. The failure management for level one consistency is much simpler because the replicas are more independent from each other.

In order to recover lost propagations after failures, each Neptune service node maintains a REDO write log for each data partition it hosts. Each log entry contains the service method name, partition ID, the request message along with an assigned *log sequence number (LSN)*. The write log consists of a committed portion and an uncommitted portion. The committed portion records those writes that are already completed while the uncommitted portion records the writes that are received but not yet completed.

Neptune assigns a static priority for each replica of a data partition. The primary is the replica with the highest priority. When a node failure is detected, for each partition that the faulty node is the primary of, the remaining replica with the highest priority is elected to become the new primary. This election algorithm is based on the classical Bully Algorithm [8] except that each replica has a priority *for each data partition* it hosts. This failover scheme also requires that the elected primary does not miss any write that has committed in the failed primary. To ensure that, before the primary executes a write locally, it has to wait until all other replicas have acknowledged the reception of its propagation. If a replica does not acknowledge in a timeout period, this replica is considered to fail due to our fail-stop assumption and thus this replica can only rejoin the service cluster after going through the recovery process described below.

When a node recovers after its failure, the underlying single-site service module first recovers its data into a consistent state. Then this node will enter Neptune's three-phase recovery process as follows:

Phase 1: Internal synchronization. The recovering node first synchronizes its write log with the underlying service module. This is done by using the registered CHECK callbacks to determine whether each write in the uncommitted log has been completed by the service module. The completed writes are merged into the committed portion of the write log and the uncompleted writes are reissued for execution.

Phase 2: Missing write recovery. In this phase, the recovering node announces its priority for each data partition it hosts. If the partition has a higher priority than the current primary, this node will bully the current primary into a secondary as soon as its priority announcement is heard. Then it contacts the deposed primary to recover the writes that it missed during its down time. For a partition that does not have a higher priority than the current primary, this node simply contacts the primary to recover the missed writes.

Phase 3: Operation resumption. After the missed writes are recovered, this recovering node resumes normal operations by publishing the services it hosts and accepting requests from the clients.

Note that if a recovering node has the highest priority for some data partitions, there will be no primary available for those partitions during phase two of the recovery. This temporary blocking of writes is essential to ensure that the recovering node can bring itself up-to-date before taking over as the new primary. We will present the experimental study for this behavior in Section 5.3. We also want to emphasize that a catastrophic failure that causes all replicas for a certain partition to fail requires special attention. No replica can successfully complete phase two recovery after such a failure because there is no pre-existing primary in the system to recover missed writes. In this case, the replica with newest version needs to be manually brought up as the primary then all other replicas can proceed the standard three-phase recovery.

Before concluding our failure recovery model, we describe a possible CHECK callback support provided by the service module. We require the Neptune server module to pass the LSN with each request to the service instance. Then the service instance fulfills the request and records this LSN on persistent storage. When the CHECK callback is invoked with an LSN during a recovery, the service module compares it with the LSN of the latest completed service access and returns appropriately. As we mentioned in Section 2.2, Neptune provides atomic execution through failures only if the underlying service module can ensure atomicity on single-site service accesses. Such support can ensure the service access and the recording of LSN take place as an atomic action.

4 System and Service Implementations

We have implemented and deployed a prototype Neptune infrastructure on a Linux cluster. The node workload is acquired through the Linux `/proc` file system. The publish/subscribe channel is implemented using IP multicast. Each multicast message contains the service announcement and node runtime workload. We try to limit the size of each multicast packet to be within an Ethernet *maximum transmission unit (MTU)* in order to minimize the multicast overhead. We let each node send the multicast message once every second and all the soft states expire in five seconds. That means a faulty node will be detected when five of its multicast messages are not heard in a row. These numbers are rather empirical and we never observed a false failure detection in practice. This is in some degree due to the fact that each node has two network interface cards, which allows us to separate the multicast traffic from other service traffic.

Each Neptune server module could be configured to run service instances as either threads or processes. The server module also keeps a growable process/thread pool and a waiting queue. When the upper limit for the growable pool is reached, subsequent requests will be queued. This scheme allows the Neptune server module to gracefully handle spikes in the request volume while maintaining a desirable level of concurrency.

We have deployed three network services in the Neptune infrastructure. The first service is *on-line discussion group*, which handles three types of requests for each discussion topic: viewing the list of message headers (`ViewHeaders`), viewing the content of a message (`ViewMsg`), and adding a new message (`AddMsg`). Both `ViewHeaders` and `ViewMsg` are read-only requests. The messages are maintained and displayed in a hierarchical format according to the reply-to relationships among them. The discussion group uses MySQL database to store and retrieve messages and topics.

The second service is a prototype *auction* service, which is also implemented on MySQL database. The auction service supports five types of requests: viewing the list of categories (`ViewCategories`), viewing the available items in an auction category (`ViewCategory`), viewing the information about a specific item (`ViewItem`), adding a new item for auction (`AddItem`), and bidding for an item (`BidItem`). Level three consistency with proper staleness bound and progressive version delivery is desirable for this service in order to prevent auction users from seeing declining bidding prices.

Our final study was a *persistent cache* service, which supports two service methods: storing a key/data pair into the persistent cache (`CacheUpdate`) and retrieving the data for a given key (`CacheLookup`). The persistent cache uses an MD5 encoding based hashing function to map the key space into a set of buckets. Each bucket initially occupies one disk block (1024 bytes) in the persistent storage and it may acquire more blocks in the case of overflow. We use `mmap()` utilities to keep an in-memory reference to the disk data and we purge the updates and the corresponding LSN into the disk at every tenth `CacheUpdate` invocation. The LSN is used to support the CHECK callback that we discussed in Section 3.2. The persistent cache is most likely an internal service, which provides a scalable and reliable data store for other services. We used level two consistency for this service, which allows high throughput with intermittent false cache misses. A similar strategy was adopted in an earlier study on Web cache clustering [13].

We note that MySQL database does not have full-fledged transactional support, but its latest version supports "atomic operations", which is enough for Neptune to provide cluster-wide atomicity. On the other hand, our current persistent cache is built on a regular file system without atomic recovery support. However, we believe such a setting is sufficient for illustrative purposes.

5 System Evaluations

Our experimental studies are focused on performance-scalability, availability, and consistency levels of Neptune cluster services. All experiments described here were conducted on a rack-mounted Linux cluster with around 40 nodes. The nodes we used in the experiments include 16 dual 400 Mhz Pentium II processors, and 4 quad 500 Mhz Pentium II Xeon processors, all of which contains 1 GBytes memory. Each node runs Linux 2.2.15 and has two 100 Mb/s Ethernet interfaces. The cluster is connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth.

Even though all the services rely on protocol gateways to reach end clients, the performance between protocol gateways and end clients is out of the scope of this paper. Our experiments are instead focused on studying the performance between clients and services inside a Neptune service cluster. During the evaluation, the services were hosted on dual-Pentium IIs and we ran testing clients on quad-Xeons. MySQL 3.23.22-Beta was used as the service database.

We used synthetic workloads in all the evaluations. Two types of workloads were generated for this purpose: 1) Balanced workloads where service requests are evenly distributed among data partitions were used to measure the best case scalability. 2) Skewed workloads, in comparison, were used to measure the system performance when some particular partitions draw an disproportional number of service requests. We measured the maximum system throughput when more than 98% of client requests were successfully completed within two seconds. Our testing clients attempted to saturate the system by probing the maximum throughput. In the first phase, they doubled their request sending rate until 2% of the requests failed to complete in 2 seconds. Then they adjusted the sending rates in smaller steps and resumed probing.

Our experience of using Linux as a platform to build large-scale network services has been largely positive. However, we do observe intermittent kernel crashes under some network-intensive workload. Another problem we had during our experiments is that the system only allows around 4000 ports to stay in the TCP TIME_WAIT state at a given time. This causes our testing clients to perform abnormally in some large testing configurations, which forces us to use more machines to run testing clients than otherwise needed. We have recently ported Neptune to Solaris and we plan to conduct further studies on the impact of different OS networking kernels.

The rest of this section is organized as follows. Section 5.1 and Section 5.2 present the system performance under balanced and skewed workload. Section 5.3 illustrates the system behavior during failure recoveries. The discussion group service is used in all the above experiments. Section 5.4 presents the performance of the auction and persistent cache service.

5.1 Scalability under Balanced Workload

We use the discussion group to study the system scalability under balanced workload. In this evaluation, we studied the performance impact when varying the replication degree, the number of service nodes, the write percentage, and consistency levels. The write percentage is the percentage of writes in all requests and it is usually small for discussion group services. However, we are also interested in assessing the system performance under high write percentage, which allows us to predict the system behavior for services with more frequent writes. We present the results under two write per-

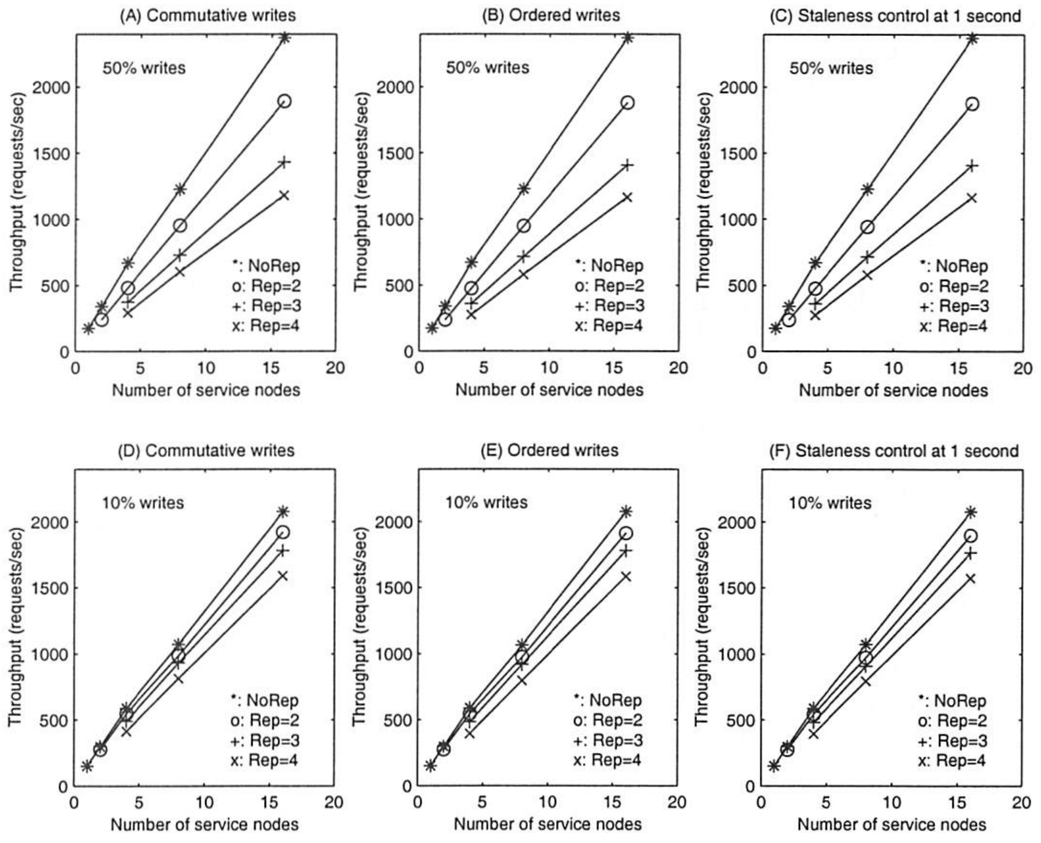


Figure 3: Scalability of discussion group service under balanced workload.

centages: 10% and 50%. In addition, we measured all three consistency levels in this study. Level one consistency requires writes to be commutative and thus, we used a variation of the original service implementation to facilitate it. For the purpose of performance comparison with other consistency levels, we kept the changes to be minimum. For level three consistency, we chose one second as the staleness bound. We also noticed that the performance of level three consistency is affected by the request rate of individual clients. This is because a higher request rate from each client means a higher chance that a read has to be forwarded to the primary node to fulfill progressive version control, which in turn restricts the system load balancing capabilities. We recognized that most Web users spend at least several seconds between consecutive requests. Thus we chose one request per second as the client request rate in this evaluation to measure the worst case impact.

The number of discussion groups in our synthetic workload was 400 times the number of service nodes. Those groups were in turn divided into 64 partitions. These partitions and their replicas were evenly distributed across service nodes. Each request was sent to a discussion group chosen according to an even distribution. The distribution of different requests (`AddMsg`, `ViewHeaders` and `ViewMsg`) was determined based on the write percentage.

Figure 3 shows the scalability of discussion group service with three consistency levels and two write percentages (10% and 50%). Each sub-figure illustrates the system performance under no replication (`NoRep`) and replication degrees of two, three and four. The `NoRep` performance is acquired through running a stripped down version of Neptune which does not contain any replication overhead except logging. The single node performance under no replication is 152 requests/second for 10% writes and 175 requests/second for 50% writes. We can use them as an estimation for the basic service overhead. Notice that a read is more costly than a write because `ViewHeaders` displays the message headers in a hierarchical format according to the reply-to relationships, which may invoke some expensive SQL queries.

We can draw the following conclusions based on the results in Figure 3: 1) When the number of service nodes increases, the throughput steadily scales across all replication degrees. 2) Service replication comes with an overhead because every write has to be executed more than once. Not surprisingly, this overhead is more prominent under higher write percentage. In general, a non-replicated service performs twice as fast as its counterpart with a replication degree of four at 50% writes. However, Section 5.2 shows that replicated services can outperform non-replicated services under skewed workloads due to better load balancing. 3) All three consistency levels perform very closely under balanced workload. This means level one consistency does not provide a significant performance advantage and a staleness control does not incur significant overhead either. We recognize that higher levels of consistency result in more restrictions on Neptune client module's load balancing capability. However, those restrictions inflict very little performance impact for balanced workload.

5.2 Impact of Workload Imbalance

This section studies the performance impact of workload imbalance. Each skewed workload in this study consists of requests that are chosen from a set of partitions according to a Zipf distribution. Each workload is also labeled with a *workload imbalance factor*, which indicates the proportion of the requests that are directed to the most popular partition. For a service with 64 partitions, a workload with an imbalance factor of 1/64 is completely balanced. A workload with an imbalance factor of 1 is the other extremity in which all requests are directed to one single partition. Again, we use the discussion group service in this evaluation.

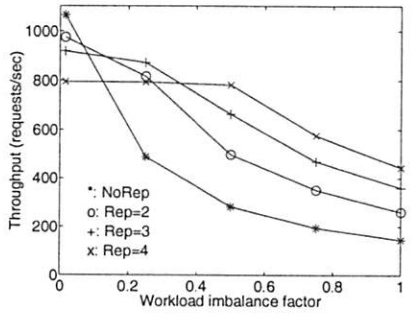


Figure 4: Impact of workload imbalance on the replication degrees with 8 service nodes.

Figure 4 shows the impact of workload imbalance on services with different replication degree. The 10% write percentage, level two consistency, and eight service nodes were used in this experiment. We see that even though service replication carries an overhead under balanced workload (imbalance factor = 1/64), replicated services can outperform non-replicated ones under skewed workload. Specifically, under the workload that directs all requests to one single partition, the service with a replication degree of four performs almost three times as fast as its non-replicated counterpart. This is because service replication provides better load-sharing by spreading hot-spots over several service nodes, which completely amortizes the overhead of extra writes in achieving the replica consistency.

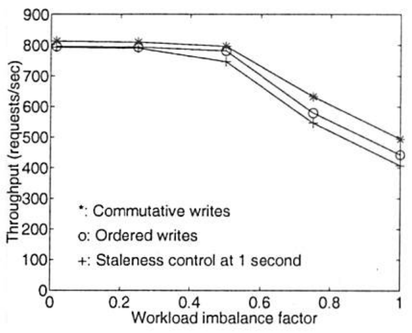


Figure 5: Impact of workload imbalance on consistency levels with 8 service nodes.

We learned from Section 5.1 that all three consistency levels perform very closely under balanced workload. Figure 5 illustrates the impact of workload imbalance on different consistency levels. The 10% write percentage, a replication degree of four, and eight service nodes were used in this experiment. The performance difference among three consistency levels becomes slightly more prominent when the workload imbalance factor increases. Specifically under the workload that directs all requests to one single partition, level one consistency yields 12% better performance than level two consistency, which in turn performs 9% faster than level three consistency with staleness control at one second. Based on these results, we learned that: 1) The freedom of directing writes to any replica in level one consistency only yields moderate performance advantage. 2) Our staleness control scheme carries an insignificant overhead even though it appears slightly larger for skewed workload.

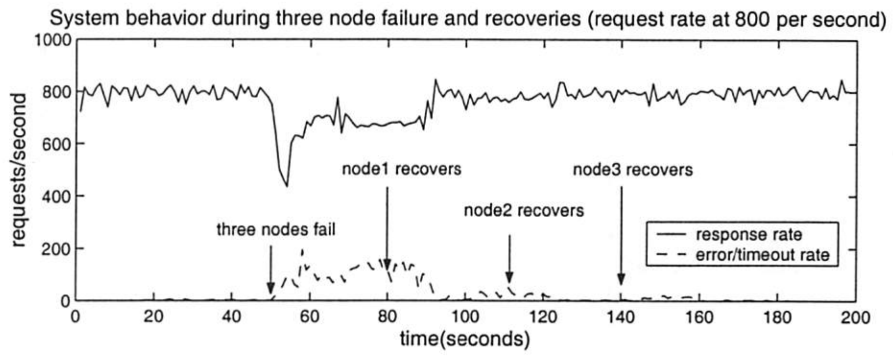


Figure 6: Behavior of the discussion group service during three node failure and recoveries. Eight service nodes, level two consistency, and a replication degree of four were used in this experiment.

5.3 System Behavior during Failure Recoveries

Figure 6 depicts the behavior of a Neptune-enabled discussion group service during three node failures in a 200-second period. Eight service nodes, level two consistency, and a replication degree of four were used in the experiments. Three service nodes fail simultaneously at time 50. Node 1 recovers 30 seconds later. Node 2 recovers at time 110 and node 3 recovers at time 140. It is worth mentioning that a recovery in 30 seconds is fairly common for component failures. System crashes usually take longer to recover, but operating systems like Linux could be configured to automatically reboot a few seconds after kernel panic. We observe that the system throughput goes down during the failure period. And we also observe a tail of errors and timeouts trailing each node recovery. This is caused by the lost of primary and the overhead of synchronizing lost updates during the recovery as discussed in Section 3.2. However, the service quickly stabilizes and resumes normal operations.

5.4 Auction and Persistent Cache

In this section, we present the performance of the Neptune-enabled auction and persistent cache service. We analyzed the data published by `eBay` about the requests they received from May 29 to June 9, 1999. Excluding the requests for embedded images, we estimate that about 10% of the requests were for bidding, and 3% were for adding new items. More information about this analysis can be found in our earlier study on dynamic web caching [24]. We used the above statistical information in designing our test workload. We chose the number of auction categories to be 400 times the number of service nodes. Those categories were in turn divided into 64 partitions. Each request was made for an auction category selected from a population according to an even distribution. We chose level three consistency with staleness control at one second in this experiment.

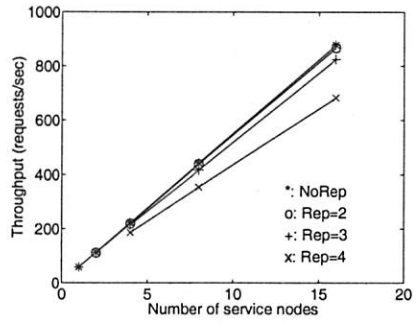


Figure 7: Performance of auction on Neptune.

Figure 7 shows the performance of a Neptune-enabled auction service. Its absolute performance is slower than that of the discussion group because the auction service involves extra overhead in authentication and user account maintenance. In general the results match the performance of the discussion group with 10% writes in Section 5.1. However, we do observe that the replication overhead is smaller for the auction service. The reason is that the tradeoff between the read load sharing and extra write overhead for service replication depends on the cost ratio between a read and a write. For the auction service most writes are bidding requests which incur very little overhead by themselves.

Figure 8 illustrates the performance of the persistent cache service. Level two consistency and 10% write percentage were used in the experiment. The results show large replication overhead caused by extra writes. This is because `CacheUpdate` may cause costly disk accesses

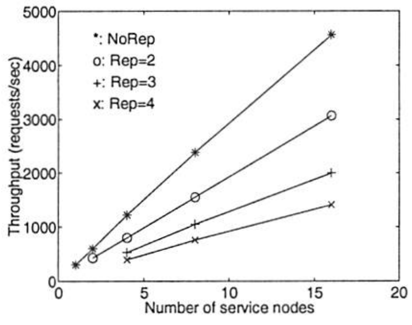


Figure 8: Performance of persistent cache on Neptune.

while `CacheLookup` can usually be fulfilled with in-memory data.

6 Related Work

Our work is in large part motivated by the TACC and MultiSpace projects [7, 12] with a focus on providing infrastructural support for data replication and service clustering. It should be noted that replication support for cluster-based network services is a wide topic. Previous research has studied the issues of persistent data management for distributed data structures [11] and optimistic replication for Internet services [18]. Our work complements these studies by providing infrastructural software support for clustering stand-alone service modules with the capability of accommodating various underlying data management solutions and integrating different consistency models.

The earlier analysis by Gray et al. shows that the synchronous replication based on eager update propagations leads to high deadlock rates [9]. A recent study by Anderson et al. confirms this using simulations [3]. The asynchronous replication based on lazy propagations has been used in Bayou [15]. Adya et al. have studied lazy replication with a type of lazy consistency in which server data is replicated in a client cache [1]. The serializability for lazy propagations with the primary-copy method is further studied by a few other research groups [3, 5] and they address causal dependence when accessing multiple objects. The most recent work by Yu and Vahdat provides a tunable framework to exploit the tradeoff among availability, consistency, and performance [22]. These studies are mainly targeted on loosely-coupled distributed services, in which communication latency is relatively high and unbounded. In comparison, our contribution focuses on time-based staleness control by taking advantage of low latency and high throughput system area networks.

Commercial database systems from Oracle, Sybase and IBM support lazy updates for data replication and they rely on user-specified rules to resolve conflicts. Neptune differs from those systems by taking advantage of the inherently partitionable property of most Internet services. As a result, Neptune's consistency model is built with respect to single data partition, which enables Neptune to deliver highly consistent views to clients without losing performance and availability. Nevertheless, Neptune's design on communication schemes and failure recovery model benefits greatly from previous work on transactional RPC and transaction processing systems [20, 10].

Providing reliable services in the library level is addressed in the SunSCALR project [19]. Their work relies on IP failover to provide failure detection and automatic reconfiguration. The Microsoft cluster service (MSCS) [21] offers an execution environment where off-the-shelf server applications can operate reliably on an NT cluster. These studies do not address persistent data replication.

7 Concluding Remarks

Our work is targeted at aggregating and replicating partitionable network services in a cluster environment. The main contributions are the development of a scalable and highly available clustering infrastructure with replication support and the proposal of a weak replica consistency model with staleness control at its highest level. In addition, our clustering and replication infrastructure is capable of supporting application-level services built upon a heterogeneous set of databases, file systems, or other data management systems.

Service replication increases availability, however, it may compromise the throughput of applications with frequent writes because of the consistency management overhead. Our experiments show that Neptune's highest consistency scheme with staleness control can still deliver scalable performance with insignificant overhead. This advantage is gained by focusing on the consistency for a single data partition. In terms of service guarantees, our level three consistency ensures that client accesses are serviced progressively within a specified soft staleness bound, which is sufficient for many Internet services. In that sense, a strong consistency model, which allows clients to always get the latest version with the cost of degraded throughput, may not be necessary in many cases. Nevertheless, we plan to further investigate the incorporation of stronger consistency models in the future.

Acknowledgment. This work was supported in part by NSF CCR-9702640, ACIR-0082666 and 0086061. We would like to thank Amr El Abbadi, Hong Tang, and the anonymous referees for their valuable comments.

References

[1] A. Adya and B. Liskov. Lazy Consistency Using Loosely Synchronized Clocks. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, pages 73–82, August 1997.

[2] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic Algorithms in Replicated Databases. In *Proc. of the 16th Symposium on Principles of Database Systems*, pages 161–172, Montreal, Canada, May 1997.

[3] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, Consistency, and Practicality: Are These Mutually Exclusive? In *Proc. of 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 484–495, Seattle, WA, June 1998.

[4] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proc. of the IEEE Intl. Symposium on Parallel Processing*, pages 850–856, April 1996.

[5] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In *Proc. of the 12th Intl. Conf. on Data Engineering*, pages 469–476, New Orleans, Louisiana, February 1996.

[6] A. Fox and E. A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proc. of HotOS-VII*, Rio Rico, AZ, March 1999.

[7] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint Malo, October 1997.

[8] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Trans. on Computers*, 31:48–59, January 1982.

[9] J. Gray, P. Helland, P. O'Neil, , and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, California, 1993.

[11] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proc. of the 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[12] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. In *Proc. of the USENIX Annual Technical Conf.* Monterey, CA, 1999.

[13] V. Holmedahl, B. Smith, and T. Yang. Cooperative Caching of Dynamic Content on a Distributed Web Server. In *Proc. of the Seventh IEEE Intl. Symposium on High Performance Distributed Computing*, July 1998.

[14] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of the ACM 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.

[15] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 288–301, Saint Malo, France, October 1997.

[16] S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proc. of ACM SIGCOMM'99*, pages 15–25, Cambridge, Massachusetts, September 1999.

[17] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability, and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, December 1999.

[18] Y. Saito and H. Levy. Optimistic Replication for Internet Data Services. In *Proc. of the 14th Intl. Conf. on Distributed Computing*, October 2000.

[19] A. Singhai, S.-B. Lim, and S. R. Radia. The SunSCALR Framework for Internet Servers. In *Proc. of the 28th Intl. Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998.

[20] BEA Systems. Tuxedo Transaction Application Server White Papers. http://www.bea.com/products/tuxedo/papers.html.

[21] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray. The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High-Availability and Scalability. In *Proc. of the 28th Intl. Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998.

[22] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of the 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[23] H. Zhu, H. Tang, and T. Yang. Demand-driven Service Differentiation for Cluster-based Network Servers. In *Proc. of IEEE INFOCOM'2001*, Anchorage, AK, April 2001.

[24] H. Zhu and T. Yang. Class-based Cache Management for Dynamic Web Contents. In *Proc. of IEEE INFOCOM'2001*, Anchorage, AK, April 2001.

System Support for Scalable, Reliable and Highly Manageable Web Hosting Service

Mon-Yen Luo and Chu-Sing Yang
Department of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C

Abstract

This paper presents the architecture and some key mechanisms of an integrated framework for providing a reliable and highly manageable Web hosting service on a scalable server cluster. We devise a novel idea termed "*URL Formalization*" and a corresponding data structure, which provide a scalable solution to the request distribution in the system. We exploit the advantages of Java to implement a management system to providing a highly manageable system. Our system supports a higher level of services reliability than other server cluster systems. The result of performance evaluation on the proposed system shows that the system is low-cost and effective.

1. Introduction

With the popularity of the Internet and the World Wide Web, the desire for using the Web to serve business transactions is increasing at an amazing rate. A successful Web site has become increasingly essential to the business community. However, constructing a successful Web site must cope with many challenging problems. First, a web site must be able to service thousands of simultaneous client requests and scale to rapidly growing user population. Furthermore, rapid response and 24-by-7 availability are mandatory requirements for a Web site as it competes for offering users the best "surfing" experience. As a result, Web hosting service is soaring as companies turn to service providers to handle these challenges, avoid infrastructure costs, and deal with staffing shortages. The Web hosting service providers offer system resources (e.g., bandwidth to the Internet, disks, processors, memory, etc.) to store and provide Web access to documents from individuals, institutions, and companies who lack the resource or the expertise to maintain a Web site.

Web server cluster [1] is a popular approach used in a shared Web hosting infrastructure as a way to create scalable and highly available solutions. However, hosting a variety of contents (i.e., documents, Web pages, resources, etc., generally called content in the Internet parlance) from different customers on such a distributed server system faces new design and management problems and requires new solutions. This paper describes the research work we are pursuing for constructing an integrated framework to address the challenges faced by hosting Web content on a server cluster environment. Our system has a variety of design goals, however, this paper focus on addressing the following two major problems:

- **Content management**
  How to place and manage hosting content in a clustered server will become a very important but challenging problem. You may imagine how tedious and difficult if you have 500 different sites with a variety of content types, and you try to place and manage these contents on a server cluster with 60 nodes. Thus, developing a management system to automate management operations and provide a logical view of a monolithic system is extremely important. In particular, such clustered servers tend to be more heterogeneous because they generally grow incrementally as needed. This is an important advantage of clustered servers so that they can scale gracefully with offered load, preserving previous investments in hardware and software. Unfortunately, this advantage will come at the cost of greatly increased management complexity.
- **Service reliability**
  The existing clustered servers merely provide high availability, but offer no guarantees about fault resilience for the service. If one server node fails, most of the existing clustered systems can mask this failure and then reconfigure the system. However, any requests in progress on the failed server will fail. This will be unacceptable to the customers who conduct E-transaction services on the Web. In addition to detecting and masking the failed nodes, an ideal fault-tolerant server should enable the ongoing requests on the failed node to be smoothly migrated and then recovered on another working node. However, this is a challenging problem, and none of the existing clustering schemes could efficiently support this capability.

Basically, our system consists of a request distributing mechanism and a Java-based management system. The distributing mechanism presents a single entry point to the external world, and manages the allocation of incoming requests to server nodes. To effectively support request distribution in the shared hosting environment, we devised a novel idea termed "*URL Formalization*" and a corresponding data structure. In addition, we also augment the distributing mechanism with a novel mechanism termed *request migration*, which transparently enables request migration and recovery in the Web server cluster in the presence of server overload or failure. The Java-based management system could relieve the administrator's burden on managing the complex system.

The rest of this paper is organized as follows. In section 2, we will describe our distributing mechanism and how it can provide a scalable solution to the request distribution in our system. The section 3 presents how we exploit several advantages of Java to implement a management system to provide a highly manageable system. In section 4, we describe how a higher reliability can be achieved by our system. We present the result of performance evaluation on the prototype system in section 5. We discuss the advantages and possible arguments of our system in section 6, and then draw conclusions in section 7.

2. Distributing Mechanism

Given a clustered server, some distributing mechanism is needed to dispatch and route the incoming requests to the server best suited to respond. In this section, we describe the design and implementation details of our distributing mechanism.

2.1 Content-Aware Routing

Over the past few years, numerous distributing mechanisms have been proposed. These schemes can be classified into the following categories: client-side approach [2,3], DNS based approach [4,5], TCP connection routing [6,7], HTTP redirection [8], and content-based routing [9,10,11]. We think that the content-based routing mechanism is the best choice to the Web hosting environment, because the other schemes only can perform request routing based on some simple criteria (e.g., number of ongoing connections). As Web services became more sophisticated, such simple routing schemes are no longer sufficient. New services such as online retailing began to use multiple tiers of servers for content and transaction processing, posing more requirements and challenges to the routing mechanisms. The content-aware routing mechanism can offer many potential benefits [11], such as sophisticated load balancing, QoS support, session integrity, flexibility in content deployment, etc.

The distributing mechanism of the proposed system is based on our previous work [11] on implementing a content-based routing mechanism. We briefly describe the operation of the content-based routing mechanism as follows; the detailed description is given in [11]. The dispatcher node that executes the routing mechanism pre-forks a number of persistent connections [12,13] to the back-end nodes, and then allocates system resources by dispatching client requests on these trunks. When a client tries to retrieve a specific content, the client-side browser first needs to create a TCP connection. The incoming TCP connection requests are acknowledged and handled at the dispatcher until the client sends packets conveying the HTTP request, which contains the URL (specifies the content it is asking for) and other HTTP client header information (e.g., Host, cookie etc.). At that point, the dispatcher looks into the HTTP header to make decision on how to route the request. When the dispatcher selects a server that is best suited to this request, it then chooses an idle pre-forked connection from the available connection list of the target server. The dispatcher then stores related information about the selected connection in an internal data structure termed "mapping table", binding the user connection to the pre-forked connection. After the connection binding is determined, the dispatcher handles the consequent packets by changing each packet's IP and TCP headers for seamlessly relaying the packet between the user connection and the pre-forked connection, so that the client and the server can transparently receive and recognize these packets.

In this paper, we further point out that such a design can enable a new capability: *request migration*. The request can be migrated to another node in the presence of server overload or server failure, which can control the resource allocation in a fine granularity and enhance service reliability, respectively. In this paper, we focus on the aspect in terms of service reliability. To support high reliability, we think the server cluster should be augmented to include two important capabilities: checkpointing and failover. That is, some intermediate state of user requests should be logged periodically by the checkpointing mechanism. If one server fails, the failover mechanism should enable the ongoing requests on the failed node to be continued processing with a valid intermediate state in another working node. Although the two techniques has been investigated and are well known in the research area of fault tolerance, implementing these techniques in the distributed web server still poses many new challenges.

First, the cost is very expensive if we log every incoming request for checkpointing. In a Web hosting system, not all content is equally important to the client and the service provider. Some of the hosting contents cannot tolerate service disruptions because of their importance or cost. With the content-aware routing capability, the distributing mechanism can differentiate the important requests (e.g., requests for mission-critical services, or requests for content owned by important customers) from regular Web surfing requests. Second, how to recover a Web request of a failed server to continue execution in another working node is a challenging problem. In particular, such recovery mechanism should be user-transparent and smooth. In the section 4, we will describe our design that provides an elegant solution to this problem.

2.2 Content-Aware Intelligence

With the above mechanism, the next question is how to build the content-aware intelligence into the dispatcher for making routing decision. To address this, we should have answers to the following questions:

- •What kind of information do we need?
- •How can we put the related information into the dispatcher?
- •How can the dispatcher perfom content-aware routing based on these information?

To the first two questions, we devised an internal data structure termed *URL table* to hold the content-related information that enables the dispatcher to make intelligent routing decisions. The possible information includes content size, type, priority, which nodes possess the content, etc. We argue that the URL table should model the hierarchical structure of the Web content. Such an argument is based on the observation that people generally organize content using a directory-based hierarchical structure. The files in the same directory usually possess the same properties. For example, the files underneath the /CGI-bin/ directory generally are CGI scripts for generating dynamic content.

Consequently, we implemented the URL table as a multi-level hash tree, in which each level corresponds to a level in the content tree and each node represent a file or directory. Basically, each item (file or directory) of content in a Web site should have a record corresponding to it in the URL table. However, to reduce the search time and the size of the table, our URL table uses a "wildcard" mechanism to specify a set of items that own the same properties. For example, if all items underneath the sub-directory "/html/" are all hosted in the same nodes and have the same content type, only the entry "/html/" exists in the URL table. If the dispatcher intends to search the URL table to retrieve the information about a URL "/html/misc.html", it can get the information from the record "/html" in the table by just one level search. Otherwise, we also implemented a mechanism to cache recently accessed entries, which is a proven technique for searching speedup. The URL table generally is self-generated, maintained, and managed by the management system (see next section) via parsing the content tree. The administrator also can configure the URL table if necessary.

Our previous experience [11] has taught us that to address the third question is a more thorny challenge. To perform content-based routing, the dispatcher should look into the HTTP header of each request. However, the HTTP header is composed of variable-length strings. Therefore, performing content-based routing implies that some kind of string search and matching algorithm is required. It is well known that such operations are time consuming. Our experience showed that the system performance would be severely degraded if we implement some string parsing functions in the dispatcher. Some vendors also made a similar observation [14], which indicated that you will loss 7/8ths of your Web switch's performance if you turn on its URL parsing function.

The above problem will be more serious in the shared Web hosting environment. We consider the following URL: *http://www.foo.com/sports/football/* as an example for explanation. This URL identifies that the content in the directory "/sports/football/" on the host "www.foo.com" can be accessed via HTTP protocol. A client wishing to retrieve this resource should create a TCP connection to the server and send a HTTP request including a request line [13] in its header:

Get /sports/football/ HTTP 1.1

, followed by the remainder of the request. The Host name of the URL will be transmitted in a Host header field of the entity-header fields [13], followed by the request line. Because all Web sites in the shared hosting environment are publicized by the same IP address to the external world, the host field is required to identify which Web site the requests is for. This implies that the dispatcher needs to look deeper in the HTTP header (not just the request line) to find the host field. As the HTTP header is composed of variable-length strings, parsing the header to retrieve such information will be a considerable burden.

To solve this problem, we devised a novel mechanism termed *URL Formalization*. Our approach is to make every directory and file of the Web content have a formalized expression. In our system, all Web objects originally reside on a reliable "home server", which is also the place for the customers to upload their content. The document stored on the home server also serves a permanent copy for consistency and robustness. Before these web objects are placed to the server system, a program will convert the original name of every directory and file into a fixed length and formatted name. Then, the program will parse all html files and script files that generate dynamic content, and modify the embedded hyperlinks to conform to the new name. In addition, the new path name of each link will be converted to a composition of the original path names under its domain name. Finally, the content is placed to the server nodes as the converted name. But, they also have the original name as an aliasing name.

For example, if an embedded link points to the above URL, the link should be converted to "/preamble/5967/1019/2048/". The name "www.foo.com", "sports", and "football" are converted to a formalized name 5967, 1019, and 2048 respectively. The preamble is a "magic number", which is designed to indicate that the following is a formalized URL. This also implies that the name of the first level directory of each server node is "/preamble", and the hosted content is placed under the directory. The design of the preamble number is important, because we should enable the dispatcher to know whether the URL of a request is in normal form or formalized form. The operations of parsing and reconstructing the HTML files and the script files are pre-computed offline. Thus, these operations do not impose any performance penalty on regular operations of the server system.

In the URL formalization scheme, the request line of the HTTP header in the above example will be:

Get /preamble/5967/1019/2048/ HTTP 1.1

The major advantage of such a design is to convert user-friendly names to routing-friendly names. In other words, our fixed-length and formalized names are easier for dispatcher to process. We even can implement the routing function in hardware for performance boosting. In addition, the artful design of placing the host name in the first level of the path name can further speed up the routing decision. The dispatcher can quickly identify that the incoming request is for which Web site, rather than parse the entire HTTP header to find out the host field. Combined with the well-designed URL table, the dispatcher can quickly retrieve related information to make routing decision.

The design of the URL formalization is based on the following observations. Generally, the reason for using the variable-length string to name a file or directory is just because it is mnemonic, thereby making it easier for humans to remember. However, in most cases an HTTP request is issued when the browser follows a link: either explicitly, when the user clicks on an anchor, or implicitly, via an embedded image or object. That is, most URLs are invisible to the users, i.e., they do not care about what name it has. Consequently, we can convert the original name to a formalized form in the manner of user transparency. However, in the relatively infrequent case where users occasionally load Web pages by typing a URL directly. That is why the magic number as a preamble is necessary, so that the dispatcher can distinguish the regular URL from the formalized URL.

3. Java-based Management System

In this section, we first consider issues that arise in designing a management system for hosting service in server cluster environment. Following this, we describe our technical design and implementation.

3.1 Analysis

Placing and managing hosting content in a server cluster environment is not a trivial job; it needs to take some important factors into consideration. First, the hosting content might be as varied as static Web pages, dynamic content (e.g., generated by a CGI script), or multimedia data such as streaming audio or video. Different contents have different requirements in terms of system resources. For example, requests for executing CGI scripts normally require much more computing resources than static file retrieval requests [15]. It is clear that some nodes with slow processors are not suitable for providing CPU-intensive dynamic content. Second, not all content is equally important to the service provider. The variety of content from different customers means that the expectations and requirements on the quality of service differ. In addition, the amount of money each customer is willing to spend and can afford to pay differs. The administrator needs to be able to place different content on different nodes for meeting their performance requirements, or to exert explicit control over resource allocation according to the variety of service agreement.

This problem motivated us to design a management system to ease the administrative operations and provide a logical view of a monolithic system to the system manager. We intended to construct a group of daemons running on each node, and enable them to cooperate to perform a variety of management functions. However, many problems arose when we tried to implement such an idea. The first major problem is platform heterogeneity, which arises from that we hope the proposed system is flexible enough that each server node can use any kind of hardware, operating system, and Web server software. This implies that these daemons must be capable of running on different platforms. The second considerable problem is *function heterogeneity,* which is because each server node might need different management functions due to hosting different content. The third problem is in terms of extensibility (or versioning and distribution problem). That is, the functionality of this management system cannot be extended or customized without rewriting, recompilation, re-installation, and re-instantiation of all existing daemons. In particular, each service provider may have it's own unique requirement.

3.2 Management Framework

For tackling the above problems, we decided to construct the management system using Java [16]. The management system is an extension of our previous work [17], which is an extensible framework to address the administration problem of a Web server cluster. Our management system is composed of the following four key components: *controller*, *broker*, *managelet* (composed of two words: manage and –let, it means a piece of code implementing a specific management function), and *remote console* (see figure 1).

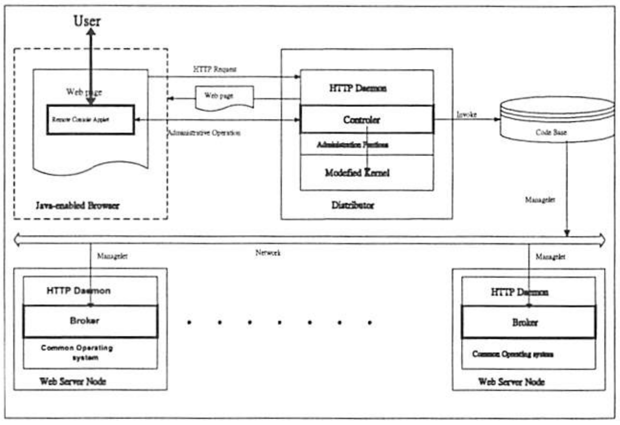


Figure 1. Overview of the Management system

The broker runs on each Web server node to perform the management functions, and monitor the status of the managed node. The broker can load Java code from the network and provides an environment for executing it. To achieve this, we implemented a customized Class Loader and Security Manager [18] as the skeleton of the broker. We also implemented a monitor thread in the broker daemon to collect the load information of the host on which it is located. The collected load information will be summarized and then sent to the dispatcher node. Each management function is implemented as a corresponding managelet, which is in the form of Java class. These managelets are stored in a reliable storage. One special daemon called *controller* will be responsible for receiving requests from the administrator, and then invokes brokers to perform the delegated tasks by dispatching the corresponding managelet to be executed on them. The remote console is a Java applet, which can be run on any Java-enabled Web browser. The administrator can download the remote console and interact with it to perform management operations.

There are two main advantages of such a design. First, implementing the daemon in Java can relieve the concerns related to heterogeneity of the target platforms. As a result, these daemons can be capable of executing on a variety of hardware architectures and operating systems. The second advantage is derived from the notion of downloaded executable content (data that contain programs that are executed upon receipt), which is a powerful feature of Java. Using this mobile code technology, we can deploy just a simple local daemon (broker) at each node, but support a variety of management functions via downloading managelets. As a result, the system can be tailored or extended to the different requirements of different system types and installations, without requiring significant redesign and coding. In the following sections, we will describe how the management system addresses some management problems of hosting content on a server cluster.

3.3 Configuration Management

We provided several administration functions on the dispatcher for the administrator to configure the server system. Examples include join/remove a server node into/from the system, read the related statistical data, etc. We defined the control interface as the boundary between these native configuration functions pertained to the dispatcher and the rest of the management system. As a result, our management system can be easily integrated with the Server-Load-Blancer from other vendors by extending the control interface to support the proprietary configuration functions of each vendor.

We then extended the functions of remote console for the administrator to configure the system. With the remote console applet, adding or removing a node is an easy job and does not require the extensive reconfiguration of all other nodes. The GUI of remote console applet supports tracking and visualization of the system's configuration and state. As a result, although the system configuration may change dynamically and availability of nodes is also subject to change, the administrator still can easily know which node is operating as a part of the system.

We also provide the monitor mechanism for the administrator to monitor the status (e.g. resource utilization) of each node, ensure the resources provided by the web site are operational, and verify the web content can be delivered normally. Otherwise, some problems that are trivially found in a single system may be hidden for a long time and difficult to be detected. The system will dispatch the managelets to execute on the managed node, collect a variety of statuses, and then report the summary to the remote console after consolidating the raw data. Reliable operational status of providing hosting service on the distributed server, unless made in such an automated way, requires significant human effort to achieve in such a complex system. In addition, the remote console also provides log and alarm functions to enable the administrator to identify security problems or other situations.

3.4 Content Management

We implemented several new management functions and system calls for the administrator to configure the URL table. The URL table is initialized by a program that parses the content tree of the Home server. We extended the remote console to visualize the URL table, which produces a single, coherent view of the Web document tree that is actually partitioned on different nodes. The remote console provides a file manager interface containing methods for inserting, deleting, and renaming files or directories. With the GUI, the administrator can easily assign different content to different servers for meeting the performance requirements of different customers. The administrator also can assign some specific content to multiple server nodes for fault tolerance or high availability. Whenever the administrator changes the document deployment, the remote console will inform the controller of these changes. The controller will change the URL table to adapt these changes, and send the managelets that perform the content management functions to propagate these changes to the whole system. As a result, although the content deployment may change dynamically, the administrator still can easily know the system's status.

We implemented several managelets to perform the content management functions. For example, one managelet is responsible to add a file into the local file system of the node that it executes. If one spare node is recruited into the server system, the managelet is sent to this node to automatically replicate some content to this node. Similarly, if the node is excluded from the system, one specific managelet is sent to offload web pages from this node. Another example is that we implement a managelet to roam in the system for checking content consistency.

4. High Reliability

In this section, we will describe how a higher reliability could be achieved by our system. Our system guarantees service reliability at tree levels. First, a status detection mechanism can detect and mask the server failures. Second, a request-failover mechanism enables an ongoing Web request to be smoothly migrated and recovered on another server node in the presence of server failure. Third, we implemented a mechanism to prevent the single-point-of-failure problem.

4.1 Failure Detection

The broker, running on each back-end server node, can also be used to provide a failure detection service for the entire system. Periodically (the interval is adaptable), the monitor thread of the broker will wake up and initiate a request to the web server running on the managed node. For minimizing the additional workload added on the managed node, such a request is designed for retrieving a small file. If the server responds normally, the broker sends an "I-am-alive" message (i.e., heartbeat message) to the dispatcher. In addition, the monitor thread also measures the response time of the issued request over each interval. If the new measured value is larger/smaller than the previous measurement, the next interval time will be increased/decreased. The increased/decreased value is proportional to the difference of the two measured values.

There are two main purposes in such a design. First, it will prevent the monitor thread from burdening the load of the web server. If the server is overloaded, the monitor thread will decrease the frequency of the probe by discovering the longer response time. Second, fine-grained load balancing can be achieved by such a design. That is, if the managed server node is overloaded, the time of interval will be lengthened. This means that the broker will increase the time of interval between this heartbeat and the next. The dispatcher keeps a counter for each server node, and such a counter will be incremented periodically. When the dispatcher receives a heartbeat from one server node, it resets the counter associated with this server node. On selecting a server for a new arriving request, the dispatcher will check the counter associated with the candidate server. If the counter exceeds a "warning value", the server node may be either overburdened or unreachable. Such a node will be skipped, and the request will automatically be allocated to the next most available server. As a result, the dispatcher could detect the overloaded node and stop dispatching new requests to it. If the counter exceeds a "dead value" (which is a higher threshold than warning value), the node will be declared dead and be temporarily removed from the server cluster, and an alarm message will be sent to the administrator. As a result, the failed server node can be detected and masked by the whole system.

4.2 Failover

In this section, we describe how an ongoing Web request can be smoothly migrated and recovered on another server node in the presence of server failure. We divide web requests into three types: requests for static content, requests for dynamic content, and requests for session-based services. We devised corresponding solution for each category. We implemented a program in the management system to parse the content tree and specify the type of each Web object in the URL table. For example, files ending with '.jpg' or 'html' are classified into the type of static content, and any URL that include 'cgi-bin' or file ending with '.php' are considered to be a dynamic content. As a result, the dispatcher can identify the type of each request via consulting the URL table, and then failover the requests of each category with the corresponding approach in case of server failure.

4.2.1 Requests for Static Content

To requests for static content, we use the following mechanism to failover a request on another node. First, the dispatcher will select a new server, and select an idle pre-forked connection connected with the target server. Then the dispatcher re-binds the client-side connection to the newly selected server-side connection. After the new connection binding is determined, the dispatcher issues a *range request* on the new server-side connection to the selected server node. From the TCP related information (i.e., ACK number, sequence number) recorded in the mapping table, the dispatcher can infer how many bytes the client has successfully received. As a result, the dispatcher can make a range request by including the *Range header* in it, specifying the desired ranges of bytes (generally starts from the last acknowledge number from the clients). Integrating with the technique of reusing pre-forked connection and seamlessly relaying packet between two TCP connections, we can smoothly recover a request on another node.

4.2.2 Requests for Dynamic Content

Some web requests are for dynamic content (hereafter, dynamic request for short), for which responses are created on demand (e.g., CGI scripts, ASP), mostly based on client-provided arguments. We cannot use the above approach to recover a dynamic request because the result of two successive requests with the same arguments may be different. The most common example is the dynamic Web pages constructed from the database. The two successive requests to the same page may be different due to the updates of the database. That means it is impossible to "seam" the results of the two requests by the range request approach described above. If we want to recover such dynamic request on another node, we should force the client to give up the data that it has received and then resubmit its request again. However, it will not be user-transparent and compatible with the existing browser.

We used the following approach to solve this problem. We made the dispatcher "store and then forward" the response of a dynamic request. In other words, the dispatcher will not relay the response to the client until it receives the complete result. Hence, if the server node fails in the middle of a dynamic request, the dispatcher will abort this connection, and then submit again the same request to another node. When it receives the complete result, it starts to reply to the client. To relieve the performance concern, we made the dispatcher to function as *reverse proxy* (or termed as *Web server accelerator* [19]). That is, the dispatcher will cache the dynamic page so that the subsequent requests for the same dynamic page can access the content from the cache instead of repeatedly invoking a program to generate the same page. We implemented the algorithm proposed in [20] to manage the cached dynamic Web pages. As a result, the system not only can solve the failure recovery problem, but also significantly benefit from this approach in terms of performance.

4.2.3 Requests for Session-based Services

A so-called session consists of a number of user interactions, i.e., the user does not browse a number of independent statically or dynamically generated pages, but is guided through a *session* controlled by a server-side program (e.g., a CGI script) associated with some shared states. For example, such a state might contain the contents of an electronic "shopping cart" (a purchase list in a shopping mall site) or a list of results from a search request. To recover a session is important because it is widely used in the E-commerce services.

However, recovering a session on another node is a more challenging problem. It requires knowledge of application-specific details such as when is the beginning of a session, internal state, intermediate parameter, when is the end of this session, and so on. It also needs a mechanism to replicate the intermediate processing-state in order to ensure the fault-tolerance of the session itself. We tackle these problems by the following mechanisms.

First of all, we model the session-based services by a state machine that consists of the following states: Start, Browse, Search, update the state, Pay, and End. The web site manager should define a session for which fault resilience or higher performance is required, by specifying some important Web pages a corresponding state. For example, the manager can define the action, "when a user adds the first item into a *shopping cart* on a specific web page," as a sign of the Start of a session; and define the action, "when user clicks the *check-out* button on a specific page", as the End of this session. The administrator could easily make such configurations via the GUI of our management system.

Such configuration information will be stored in the URL table.

As we described above, the dispatcher should consult the URL table to assign the incoming request to one of the web servers. When the dispatcher finds (here, we see again the benefit and necessity of content-aware routing mechanism) a request conveying the "start" action, it will "tag" this client and then provide the fault tolerance support for all consequent requests, until it finds a request conveying the "end" action. We design a primary-backup protocol [21] to replicate the intermediate state of a session on a backup node. When the primary server that is responsible for processing the session fails, the backup server can take over its job with the replicated state. The readers are referred to [21] for further details.

4.3 System Robustness

We noticed that the dispatcher represents a single-point-of-failure in our system, i.e., failure of the dispatcher would bring down the entire Web server. To improve the robustness of our system further, we can use multiple dispatchers to cooperate for distributing requests. In this configuration, the DNS approach [22] can be used to map different clients to different dispatchers.

We implemented a collection of daemon processes (based on the SwiFT toolkit [23,24]) that provides fault tolerance facilities on the group of dispatcher nodes, logically configured as a ring. Each dispatcher node runs the daemon process that monitors and backups its logical neighbor's state. All the dispatchers will participate in load sharing under normal operating conditions, i.e., no dispatcher is relegated to an idle hot standby status waiting for the failure of a primary dispatcher.

The dispatcher operates based on two important states: URL table and connection binding information. The URL table is a soft state that can be regenerated after the failure. In contrast, the connection binding information is a hard state that should be replicated in the backup node. Consequently, we made the primary dispatcher keeps a log of recent change of connection binding information, and periodically replicates the state change to its backup node to refresh the replicated table. If the primary fails, the backup can take over the primary's job with the replicated state.

5. System Evaluation

This section presents the results of a performance study on the prototype system. Due to space limitation, we focus the discussion on the performance experiments that evaluate the benefit of our URL formalization mechanism and URL table. We will report the performance data and a detailed analysis of the fault tolerance mechanism in [21].

5.1 Measurement Setup

We constructed the following server cluster in laboratory for performance evaluation. We used a Pentium-2 (350 MHz CPU with 128 MB memory) machine running Linux (version 2.2.12) to serve as a dispatcher. The server cluster consists of the following machines: four Pentium Pro (200 MHz CPU with 128MB) machines running Linux with Apache (version 1.2.4), and six Pentium-2 (300 MHz CPU with 128 MB memory) running Windows NT with IIS 4.0. The reason for such a software configuration is that we want to show that our mechanism can operate with any kind of operating system and server software. The Apache servers are responsible for providing static content and session-based service, and the IIS servers are responsible for providing dynamic content. We connected all these machines directly by a 3Com 3300 switch using 100Mbps full-duplex network connections.

We used 24 Pentium-2 (350 MHz CPU with 128 MB memory) machines to run the WebStone [25] benchmark for generating a synthetic workload to evaluate the proposed system. Each machine runs four WebStone client programs that emit a stream of Web requests, and measure the system response. The generated loads are varied in experiments by varying the number of WebStone clients. We also inserted the session model into the workload so that session-based service could be investigated. We implemented [21] a session generator in the WebStone client to issue session-based requests. The content hosted in the cluster system consisted of 97 Web sites (with approximately 72000 unique files of which the total size is about 1312MB).

To quantify the benefit of the new content-aware routing mechanism, we measured and compared the response time and throughput in the prototype system equipped with the new mechanism, with those in a baseline system without the proposed mechanism. The baseline system is a server cluster (with the same configuration described above) front-ended by the dispatcher implementated in our previous work [11]. The dispatcher in the baseline system needs to parse the entire HTTP header to find out the host field for making routing decision.

5.2 Results

The peak throughput of the proposed system is 3278 requests/sec. In contrast, the peak throughput of our previous system is 2365 requests/sec. At the period of peak throughput, the CPU utilization of the dispatcher equipped with the new mechanism is 52% (and the previous is 78%). The results show a significant performance improvement.

The reason for this higher performance is because of the clever design of URL formalization and its associated data structure. The dispatcher can quickly identify that the incoming request is for which Web site, rather than parse the entire HTTP header to find out the host field. Combined with the well-designed URL table, the dispatcher can quickly retrieve related information to make routing decision. To quantify the benefits of such a design, we instrumented the Linux kernel of the dispatcher to measure the latency of URL parsing and searching. We generated a heavy load (128 clients) to push the server into a prolonged overload state, and then we measured the processing time (i.e., the time of parsing a HTTP header and the time of searching the URL table to retrieve the routing information).

As we described above, the content hosted on the test system contained about 72000 Web objects. In such scale, the memory consumed by the URL table is about 1.8 Mbytes. During the peak load, the average processing time on a HTTP request is about 2.14 μsec. In contrast, the average processing time in the baseline system is about 248 μsec.

The major concern of a system equipped with some fault-tolerance mechanisms might be how much overhead is associated with these mechanisms, and if the system's performance will suffer from the additional overhead. The higher throughput of the proposed system can demonstrate that the performance concern does not exist in our system. To precisely quantify the additional overhead, we analyzed and compared the response time of a Web request in our system, with those in a baseline system without fault-resilience support. Compared with the average response time of the requests in the two configurations, we could quantify how much additional overhead will be introduced by the fault resilience mechanism.

The results of requests for static content are given in Table 1. We do not see any performance degradation introduced by the fault resilience mechanism. We also found a similar result in the performance data regarding the dynamic content. The reason of the low overhead is that we do not need to keep any additional state for recovering a request for static and dynamic content.

Request size (Kb)	**4K**	**8K**	**32K**
Our system (ms)	24.89	34.53	171.93
Baseline (ms)	23.58	32.25	170.24
Request size (Kb)	**64K**	**256K**	**1024K**
Our system (ms)	305.29	1147.42	4809.14
Baseline (ms)	308.39	1145.62	4815.17

Table 1 Overhead (Static content)

In terms of session-based requests, our protocol introduces an overhead of about 7% over the baseline system that does not offer any guarantee. The additional latency mostly comes from the need to wait for the backup node to store the processing state. Notice that the experiment was measured over a local area network, where high-speed connections are the norm, resulting in short observed response time and then large relative overhead. The overhead would be insignificant when comparing with the typical latency over the wide-area networks [26,27].

6. Discussion

In this section, we discuss the advantages of our system and possible arguments against it. We also describe the related work.

6.1 Advantages

The first advantage of our system is that we provide an innovative management system to address the management problem of deploying web hosting services on a server cluster. With the management system, hosting service providers can easily manage and maintain content on the distributed server as a single large system. The management system provides the administrator with a single system image on system management. The system can also automate many management operations that could ease the burden of system management. In addition, our management system offers a natural concurrent-problem-solving paradigm, which provides a scalable solution to some tedious management tasks in a large distributed server. For example, we implemented a managelet to retrieve some specific URL to determine the availability of the content. The administrator can configure and dispatch the managelet to several nodes to perform content-level health checks concurrently.

The second advantage of our system is that we provide a higher level of service-reliability support than other server cluster systems. While the other systems can only provide high availability by masking the server failures, our system can provide high reliability by enabling the requests on the failed node to be recovered on another working node. Such a capability is important and essential for the E-commerce providers. They are willing to pay higher fee to get the guarantee of high service reliability, since they know that service outage in today's highly competitive marketplace can mean lost revenue and lost credibility. Our content-aware routing mechanism can enable the Hosting service providers to charge for differentiated services. The dispatcher can identifies the type and importance of each request, and then provides corresponding level of reliability support.

HAWA [3] addressed the service availability problem in client side by an applet-based approach. This is a very interesting approach and has the advantage of low overhead. However, they do not address the server-side fault tolerance problem and deal with the transaction-based service. Singhai et al. [28] and Chawathe et al. [29] proposed a framework for building a highly available Internet service on the cluster system. These systems indeed could provide higher availability, however, they are not fault-tolerant or highly reliable. None of these approaches or systems addresses the issue of request migration in presence of server failure and smooth provision of service while migration occurs. Ingham et al. [30] surveyed some existing approaches for constructing a highly dependable Web server. They also pointed out the importance of providing transactional integrity and found that it is not addressed in the existing system.

The idea of content-aware routing is not new. In addition to us, a number of research projects [9,10,31] and commercial products [e.g., 32-40] also use a similar idea. However, we make the following unique contributions. First, we augment the content-aware routing with the fault-resilience capability. Second, we devised the new idea of URL formalization and a corresponding data structure. Comprehensive content-specific knowledge can be stored in the table, from self-learning by the management system or administrator's inference. Without the support of the content management system, configuring and managing the URL table will be a difficult problem. In contrast, the data structures and what kind of content-aware intelligence used in these commercial products are unclear, at least, have never been described clearly in the literatures.

Finally, the request migration technique not only can be used to provide fault resilience; it is also beneficial to system management. If we need to perform several management actions on one server node, and that might overburden this node, the system can automatically "migrate" the ongoing requests on this node to other nodes. Otherwise, if we want to temporary remove a node for maintenance, the request migration mechanism can be invoked to transfer the ongoing requests to other nodes. The same technique can also be used to solve the "flash crowd" problem. It is well known that certain event on particular web site could trigger a significant load burst that persists for hours, or even days. Examples include announcement of a new version of popular software or product, a site mentioned as the "best-site-of-the-week" on the news, or political sites during a campaign. Although the clustered server can provide compelling performance and accommodate the growth of web traffic, it could suddenly become swamped due to receiving far more requests on one node than it was originally configured to handle. The request migration mechanism can be invoked in response to server overloaded.

6.2 Possible Arguments Against our System

Someone might argue that the content management problem could be solved by a shared file system. Through the communication network, server nodes at different locations can access the content from the shared file system to serve user requests. The advantage of this approach is that we can easily manage and maintain the content with a centralized policy. However, such a design will suffer from the following problems. First, accessing data over the network file system will increase user perceived latency due to the overhead of remote-file-I/O and LAN congestion. Some specific distributed file systems (e.g., AFS [41]) that provide cache mechanism might alleviate this problem. However, it is well-known that the access patterns of WWW exhibit high skew with regard to the access frequencies of content, thus the popular files tend to occupy RAM space in all the nodes. The redundant replication of "hot" content through the RAM of all the nodes leaves much less of available RAM space for the rest of the content, leading to a worse overall system performance. Second, the shared file system approach does not take the variety of content into consideration.

6.3 Work in Progress

We are pursuing on extending and completing the functionality of the system in the following aspects. First, we are designing a load balancing mechanism to ensure an even load distribution in the system. Since the access patterns of WWW exhibit high skew, the static content-placement may lead to load imbalance. In other words, the servers that store the popular documents will get overloaded, resulting in hot spots and bad system performance. As a result, we are implementing an auto-replication facility to dynamically adjust content placement for ensuring an even load distribution. Cherkasova et al. [42,43], Narendran et al. [44], Rabinovich et al. [45,46] have done great works in a similar problem.

Second, we are investigating the issue of service Level agreement (SLA), which will enable the content owners to specify their specific requirements such as bandwidth usage, number or placement of content replicas, or required degrees of service reliability. With the proprietary request-failover mechanism, our system can offer a strong SLA (service-level agreement) on service reliability: the important customers can be promised that no user requests will be lost due to server overload or failure. We are implementing the related mechanisms to configure the management policy to meet the complex requirements of different customers.

Third, we are investigating how to support service differentiation and QoS guarantee to satisfy the customer's requirement.

7. Conclusion

Business demand is fueling the market for companies that specialize in hosting and managing other companies' Web sites. However, there are lots of actions and functions in most hosting service providers at the low end today. They must gradually move to more sophisticated services as the content of a Web site and e-business operations become more complex. In this paper, we have described the research work we are pursuing in this direction. We provide an integrated framework to construct a reliable and highly manageable Web hosting service on a scalable server cluster. We have demonstrated that the URL Formalization mechanism and a corresponding data structure can provide a scalable solution to the request distribution in the system. The management system can mask the complexity of such a distributed environment, providing a highly manageable system. A higher level of services reliability can be achieved by the proposed system. We believe that we have taken an important step toward providing a successful hosting service.

Availability

You can get more information about this system and see a demo demonstration of the remote console on our Web site: http://pds.cse.nsysu.edu.tw.

Acknowledgement

This work was supported by the National Science Council, R.O.C., under contract no. NSC 89-2622-E-110-003.

References

[1] A. Fox, S. Gribble, Y. Chawathe and E. A. Brewer, "Cluster-based scalable network services," In Proceedings of SOSP '97, October 1997.

[2] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler, "Using smart clients to build scalable services," In Proceedings of the 1997 USENIX Annual Technical Conference, January 6-10, 1997.

[3] Y. M. Wang, P. Y. Chung, C. M. Lin, and Y. Huang, "HAWA: a client-side approach to high-availability web access," In Proceedings of the Sixth International World Wide Web Conference, April 1997.

[4] R. McGrath T. Kwan and D. Reed, "NCSA's World Wide Web server: design and performance," IEEE Computer, November 1995.

[5] V. Cardellini, M. Colajanni, P.S. Yu, "DNS dispatching algorithms with state estimators for scalable Web-server clusters", World Wide Web Journal, Baltzer Science, Vol. 2, No. 3, pp. 101-113, Aug. 1999.

[6] D. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A scalable and highly available web server," In Proceedings of the COMPCON'96, February 1996.

[7] G. D. H. Hunt, G. S. Goldszmidt, R. P. King and R. Mukherjee. "Network dispatcher: a connection router for scalable Internet services," In the Proceedings of the 7th International World Wide Web Conference, April 1998.

[8] D. Andresen, T. Yang, O. Ibarra, "Towards a scalable distributed WWW server on networked workstations," Journal of Parallel and Distributed Computing, Vol 42, pp. 91-100, 1997.

[9] V. Pai, M. Aron, M. Svendsen, G. Banga, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-aware request distribution in cluster-based network servers," In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[10] A. Cohen, S. Rangarajan, and H. Slye, "On the performance of TCP splicing for URL-aware redirection," In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, October 11-14, 1999.

[11] C. S. Yang and M. Y. Luo, "Efficient support for content-based routing in Web server clusters," In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, October 11-14, 1999.

[12] J. C. Mogul, "The case for persistent-connection HTTP," In Proceedings of the SIGCOMM'95 August 1995.

[13] T. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, J. C. Mogul. Hypertext Transfer Protocol–HTTP/1.1--Draft Standard RFC 2616, June 1999.

[14] The Advantages of F5 Layer 7 Management, White paper of F5 Lab. Available at http://www.f5.com/solutions/whitepapers/layer7.html

[15] A. Iyengar, E. MacNair and T. Nguyen, "An analysis of Web server performance," In Proceedings of the GLOBECOM '97, November 1997.

[16] K. Arnold and J. Gosling. The Java Programming Language. Addison-Wesley Publishing Company, Reading, MA, 1998.

[17] C. S. Yang and M. Y. Luo, "Design and implementation of an administration system for distributed Web server," In Proceedings of the 12th USENIX Systems Administration Conference, December 1998.

[18] L. Gong. Inside Java 2 Platform Security, Addison Wesley, Reading, MA, June 1999.
[19] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias, "Design and performance of a Web server accelerator," In Proceedings of the INFOCOM'99, March 1999.
[20] J. Challenger, A. Iyengar, and P. Dantzig, "A scalable system for consistently caching dynamic Web data," In Proceedings of the INFOCOM'99, March 1999.
[21] M. Y. Luo and C. S. Yang, "Constructing Zero-loss Web Services," In Proceedings of the INFOCOM 2001, April 22-26, 2001.
[22] M. Colajanni, P.S. Yu, D.M. Dias, "Analysis of task assignment policies in scalable distributed Web-server systems", IEEE Trans. on Parallel and Distributed Systems, Vol. 9, No. 6, June 1998.
[23] Y. Huang and C. M. R. Kintala, "Software implemented fault tolerance: Technologies and experience," In Proceedings of 23rd Intl. Symposiumon Fault-Tolerant Computing, pages 2–9, Toulouse, France, June 1993.
[24] Y. Huang et.al., "NT-SwiFT:Software implemented fault tolerance on Windows NT," In Proceedings of the 2nd USENIX NT Symposium, Seattle, August 1998.
[25] WebStone, http://www.sgi.com/
[26] P. Barford and M. E. Crovella, "Measuring Web performance in the wide area," Performance Evaluation Review, August 1999.
[27] M. Kalyanakrishnan, R. K. Iyer, and J. U. Patel. "Reliability of Internet hosts: a case study from the end user's perspective," Computer Networks, 31, pp. 47-57, 1999.
[28] A. Singhai, S.-B. Lim, and S. R. Radia, "The SunSCALR framework for Internet servers," In Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, June 1998.
[29] Y. Chawathe and E. A. Brewer, "System support for scalable and fault-tolerant Internet services," In Proceedings of Middleware '98, September 1998.
[30] D. Ingham, F. Panzieri, S.K. Shrivastava, "Constructing dependable Web services,' IEEE Internet Computing, Vol.4, N. 1, January/February 2000.
[31] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, D. Saha, "Design, implementation and performance of a content-based switch," In Proceedings of the Infocom 2000, March 2000.
[32] Alteon. Alteon 180 Web Switch. http://www.alteonwebsystems.com
[33] ArrowPoint. CS-100 Switch. http://www.arrowpoint.com/
[34] Coyote Point. Equalizer. http://www.coyotepoint.com
[35] F5Labs. Big/IP. http://www.f5.com/
[36] Foundry Networks. ServerIron. http://www.foundrynet.com.
[37] Lucent. WebSwitch. http://www1.bell-labs.com/project/webswitch/default.htm
[38] HydraWeb. Hydra 5000. http://www.hydraweb.com
[39] IPivot. http://www.intel.com/network/ipivot/index.htm
[40] Resonate, http://www.resonate.com.
[41] M. Satyanarayanan. "Scalable, secure, and highly available distributed file access," IEEE Computer 23, 5 May 1990.
[42] L.Cherkasova, "FLEX: load balancing and management strategy for scalable Web hosting service," In Proceedings of the Fifth International Symposium on Computers and Communications, July 3-7, 2000.
[43] L. Cherkasova and S. Ponnekanti, "Achieving load balancing and efficient memory usage in a Web hosting service cluster," HP Laboratories Report No. HPL-2000-27, February 2000.
[44] B. Narendran, S. Rangarajan and S. Yajnik, "Data distribution algorithms for load balanced fault tolerant web access", In Proceedings of the Symposium on Reliable and Distributed Systems, October 1997.
[45] M. Rabinovich and A. Aggarwal, "RaDaR: a scalable architecture for a global Web hosting service," In Proceedings of the 8th International World Wide Web Conference, May 1999.
[46] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal, "A dynamic object replication and migration protocol for an Internet hosting service," In Proceedings of the IEEE International Conference on Distributed Computing Systems, May 1999.

Fine-Grained Failover Using Connection Migration

Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan
MIT Laboratory for Computer Science
Cambridge, MA 02139
{snoeren, dga, hari}@lcs.mit.edu

Abstract

This paper presents a set of techniques for providing fine-grained failover of long-running connections across a distributed collection of replica servers, and is especially useful for fault-tolerant and load-balanced delivery of streaming media and telephony sessions. Our system achieves connection-level failover across both local- and wide-area server replication, without requiring a front-end transport- or application-layer switch. Our approach uses recently proposed end-to-end "connection migration" mechanisms for transport protocols such as TCP, combined with a soft-state session synchronization protocol between replica servers.

The end result is a robust, fast, and fine-grained connection failover mechanism that is transparent to client applications, and largely transparent to the server applications. We describe the details of our design and Linux implementation, as well as experimental data that suggests this approach is an attractive way to engineer robust systems for distributing long-running streams; connections suffer relatively small performance degradation even when migration occurs every few seconds, and the associated server overhead is small.

1 Introduction

Ensuring a high degree of reliability and availability to an increasingly large client population is a significant challenge facing Internet content providers and server operators today. It is widely recognized that the computers serving Web content and streaming media on the Internet today do not possess the same impressive degree of reliability as other mission-critical services such as gateways and switches in the telephone network.

An effective way to engineer a reliable system out of unreliable components is to use redundancy in some form, and server replication is the way in which reliable and available services are provided on the Web today. Because most Web connections last for relatively short periods of time, the problems of load-balancing client requests and recovering from unreachable replica servers can usually be handled at the granularity of a complete connection. Indeed, this is the approach seen in several current systems that perform server selection using a front-end transport- or application-layer switch [2, 6, 11, 18, 20], or using wide-area replication for Web content distribution [1, 8].

While existing replication technologies provide adequate degrees of reliability for relatively short Web connections, streaming media and Internet telephony display substantially longer transfer lengths. Providing reliable, robust service over long connections requires the ability to rapidly transition the client to a new server from an unresponsive, overloaded, or failed server *during* a connection [16]. The requirements of these emerging applications motivate our work.

We have designed and implemented a system that achieves fine-grained, connection-level failover across both local- and wide-area server replicas, *without* a front-end transport- or application-layer switch. Thus, there are no single points of failure or potential front-end bottlenecks in our architecture. We achieve this using a soft-state session synchronization protocol between the replica servers, combined with a connection resumption mechanism enabled by recently-developed end-to-end *connection migration* mechanisms for transport protocols such as TCP [22] and SCTP [25]. The end result is a robust, fast, and fine-grained server failover mechanism that is transparent to the client application, and largely independent of the server applications. Applications that can benefit from this include long-running TCP connections (e.g., HTTP, FTP transfers), Internet telephony, and streaming media, enabling them to achieve "mid-stream failover" functionality.

Our architecture is end-to-end, with active participation by the transport stack of all parties involved in the communication. Our design is largely application-

This research was supported in part by DARPA, NSF, NTT Corporation, Intel, and IBM. Alex C. Snoeren is supported by a National Defense Science and Engineering Graduate (NDSEG) Fellowship.

independent: applications do not need to be modified to benefit from the fine-grained failover techniques. However, because we allow a server to seamlessly take over a connection from another in the middle of a data stream, there needs to be some mechanism by which the servers synchronize application-level state between themselves. While this is a hard problem in general, there are many important common cases where it is not as difficult. For example, when each client request maps directly to data from a file, our lightweight synchronization mechanism performs quite well. If content is being generated dynamically in a fashion that is not easily reproduced by another server, handoff becomes harder to accomplish without additional machinery.

We discuss several issues involved in designing a connection failover mechanism in section 2. Section 3 describes an end-to-end architecture for fine-grained server failover targeted at long-running transfers. Our TCP-based Linux implementation is described in section 4. Section 5 contains performance studies showing the effectiveness of the failover mechanism and its resilience to imperfections in the health monitoring subsystem. We conclude with a synopsis of related work in section 6 and summarize our contributions in section 7.

2 Components of a failover system

This section describes three components that a complete fine-grained failover system should provide:

(i) For any connection in progress, a method to determine when (and if) to move it to another server;
(ii) a selection process to identify a set of new server candidates; and
(iii) a mechanism to move the connection and seamlessly resume the data transfer from the new server.

2.1 Health monitoring

In general, the end-point of a connection is changed because the current server is unresponsive; this may happen because the server is overloaded, has failed, or its path to the client has become congested. The failover system needs to detect this, following which a new server can be selected and the connection appropriately moved. We call the agent in the failover system that monitors the health and load of the servers the *health monitor*.

There are several possible designs for a health monitor, and they can be broadly classified into centralized and distributed implementations. Our architecture accommodates both kinds. We believe, however, that it is better for the health monitor to be controlled by the servers than the client. It is often much harder for a client to have the requisite knowledge of the load on other servers, and putting it in control of making movement decisions may actually worsen the overall performance of the system.

The focus of this paper is not on novel mechanisms for load or health monitoring; instead, we leverage related work that has already been done in this area [3, 12, 17]. For the purposes of this paper, we assume that each server in a support group is notified of server failure by an omniscient monitor at the same time. As the experimental results in section 5 demonstrate, however, our system allows the health monitoring component to be relatively simple—*and even overly reactive*—without significantly degrading performance.

2.2 Server selection

Once the system decides that a connection should be moved to another server, it must select a new server to handle the connection. One possibility is to use a content routing system and treat this as a new request to decide which server to hand it to. Another is to have the set of relatively unloaded servers attempt to take over the connection, and arrange for exactly one of them (ideally the closest one) to succeed. Our failover architecture admits both styles of server selection.

2.3 Connection migration and resumption

Once a connection has been targeted for movement and a new server has been selected to be the new end point, the client application should seamlessly continue without noticeable disruption or incorrect behavior. This requires that the application data stream resume from exactly where it left off at the old server. To achieve this in an application-independent fashion, the transport-layer state must be moved to the new server, and the application-layer state appropriately synchronized and restarted.

There are different ways of doing this: one is a mechanism integrated with the application where the clients and servers implement a protocol that allows the server to inform the client that its communicating peer will change to a new one. Then, the client application can terminate the connections to the current server and initiate them to the new one, and retrieve the portions of the stream starting from where the previous server left off. An alternate approach, which our work enables and advocates, is an application-independent mechanism by using a secure, transport-layer connection migration mechanism. The advantage of this method is that existing applications can benefit from this without modification, while new ones do not each need to incorporate their own mechanisms to achieve these results.

3 Architecture

Our architecture preserves the end-to-end semantics of a connection across moves between servers. Rather than inserting a Web switch or similar redirecting device, we associate each connection with a subset of the servers in the system[1]. This is the connection's *support group*, the collection of servers that are collectively responsible for the correct operation of the connection. Each support group uses a soft-state synchronization protocol to distribute weakly consistent information about the connection to each server in the group. This information allows a stream to resume from the correct offset after a move.

When the health monitor determines that a connection should be moved, each of the remaining servers in the connection's support group becomes a *candidate server* for the orphaned client connection. Thus, the responsibility for providing back-up services to orphaned connections is shared equally among the other servers, and the new server is chosen from the support group in a decentralized manner by the client.

Determining the precise point of failure of a server is a difficult problem, but is fortunately one that does not need to be solved. It is sufficient to determine when the client believes the server failed. In the case of sequenced byte streams, this can be represented by the last successfully received contiguous byte as conveyed in the transport-layer acknowledgment message. Hence the new server need only elicit an acknowledgement packet from the client in order to determine the appropriate point from which to resume transmission.

The state distribution protocol periodically disseminates, for each connection, the mapping between the transport layer state and the application-level object being sent to the client (e.g., the TCP sequence number and an HTTP URL). For the remainder of this paper, we will use the term *stream mapping* to describe the task of translating the particular transport-layer sequence information into application-level object references the server can understand. The transport-layer state is moved to the new server using a secure *connection migration* mechanism [22]. Together, the techniques described above allow for the correct resynchronization of both transport-layer and application state, transparent to the client application

Figure 1 shows the basic architecture of our system for a simple configuration with two servers, A and B, in a support group. When a client wishes to retrieve an object, it is initially directed to Server A through some server selection mechanism. The connection is observed by the stream mapper, which parses the initial object request and advertises the object currently being served and the necessary stream mapping information to the rest of the support group.

[1]This subset could in fact be the entire set; partitioning the set into subsets enhances scalability.

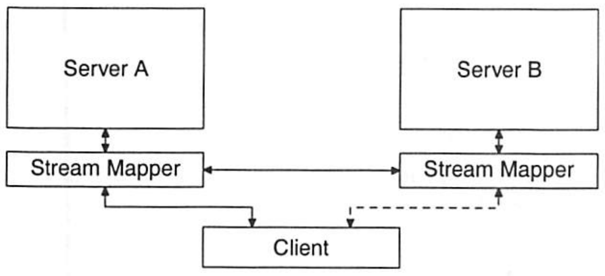


Figure 1: Failover architecture for a support group of two servers.

At some point Server B may attempt to assume control of the connection. This may be caused by the receipt of a notification from the health monitor that Server A died, or may be initiated by a load-balancing policy mechanism. Server B initiates a connection migration by contacting the client *in-band* on the same connection, using a secure transport-layer connection migration mechanism. If it receives a transport-layer acknowledgement from the client, it knows that it was the candidate server selected by the client. Because this acknowledgment notifies the server of the next expected data byte, and because the soft-state synchronization protocol in the support group has already informed Server B of the content being served, content delivery can now be resumed at the correct point. Observe that at no point was the client actively involved in the decision to migrate the connection, yet its transport-layer is fully aware that the migration took place and it even had the opportunity to pick one of several candidate servers[2]. All further client requests on the same connection are now directed to Server B, at least until the next migration event.

In the rest of this section, we describe the formation and maintenance of support groups, the soft-state synchronization protocol, and the details of the transport-layer connection migration.

3.1 Support groups

The size and distribution of support groups clearly has a large impact on the performance of our scheme. As the number of servers in a support group increases, the increased load to any one server of a death in the group decreases. Unfortunately, the communication load also

[2]If required, it is possible to provide information about the migration to the client application; we have not yet explored the ramifications of this, but plan to do so in the future.

increases, as each member of the group must advertise connection state to the others.

Since the set of candidate servers for a client whose initial server dies is bounded by the set of servers in the support group, it may be desirable to have a diversity of network locations represented in the support group to increase the chance the client has an efficient path to a candidate server. This depends greatly, however, on the effectiveness of the initial server selection mechanism, and the stability of that choice over the duration of the connection. We note that it may be desirable to limit the number of candidate servers that simultaneously attempt to contact the client in large support groups, as the implosion of migration requests may swamp the client. As a practical matter, the quadratic growth in communication required between the servers in a support group will likely limit support group size to a reasonable number.

Clearly, the choice of a live initial server is an important one, and much previous work has addressed methods to select appropriate servers in the wide area. Good servers can be identified using BGP metrics [5], network maps [1], or application-specific metrics. Similarly, clients can be directed to these servers using DNS redirection [1], HTTP redirection, BGP advertisements, or anycast [9]. Since our architecture allows connections to be handed off to any other server at any point in the connection, the ramifications of a poor initial selection are not as severe as with current schemes.

We construct support groups by generating a well-known hash of the server's IP address. By setting the number of hash buckets to n, servers are uniformly allocated into one of n support groups. This mechanism allows for the servers to be dynamically added and removed from the server pool. As each server comes on-line, it computes the hash of its interface address and then begins advertising its connections to a well-known multicast address for that hash value[3]. Every server in the support group becomes a candidate server for the connections from the new server immediately after receiving the first advertisement. Similarly, the new server begins listening to advertisements sent to the group's address, and becomes a candidate server for any connections advertised after its group membership[4].

The choice of support group membership and final server that handles a failed client should be engineered in a manner that avoids the "server implosion" problem, where all of the clients from a failed server converge on the same replacement server. We provide two mechanisms to support implosion avoidance. The first is an engineering choice: by choosing smaller support group sizes (g members per support group out of n total servers), we can limit with high probability the expected number of clients that converge on a particular server to an $O(g/n)$ fraction of the clients from the dead server. The second is by delaying connection resumption by a load-dependent factor at each server, increasing the likelihood the client is served by a less crowded server. We evaluate this technique in section 5.

3.2 Soft-state synchronization

The information advertised to the support group about each content stream can be divided into two parts. The first portion contains application-dependent information about the object being retrieved from the server, such as the HTTP URL of the client request. The second portion is the transport-layer information necessary to migrate the connection. In general, this includes the client's IP address, port number, and some protocol-specific transport layer state, such as the initial sequence number of the connection. The amount and type of transport-specific information varies from protocol to protocol. We are currently exploring a framework that describes the necessary information in a protocol-independent fashion.

The state information for a particular connection may be unavailable at some servers in the support group because the connection has not yet been announced, or because all of the periodic announcements were lost. The first failure mode affects only a small window of new connections, and can be masked as an initial connection establishment failure. We assume that connections are long-running, so the second failure mode must be guarded against more carefully. If at least one server in the support group has information about the connection, it can be re-established. We therefore need only bound the probability that no servers have fresh information about connection. It suffices to pick suitable advertisement frequencies and support group sizes (see, e.g., the analysis of a similar protocol by Raman *et al.* [19]), hence it is possible to achieve sufficient robustness in our synchronization protocol with markedly less complexity than a strongly-consistent mechanism.

3.3 Transport-layer connection migration

When attempting to resume a connection previously handled by another server in a fashion transparent to the client application, previous approaches have forged the server's IP address, making packets from the new server indistinguishable from the previous ones. This approach has two major drawbacks:

[3]The required multicast functionality may be provided at the IP layer or through an application-layer overlay.

[4]The initial request redirection mechanism must also be informed of the new server if it is to handle new client requests.

- The new server and the failed server must be co-located. Since they both use the same IP address, packets from the client will be routed to the same point in the network.
- The previous server cannot be allowed to return to the network at the same IP address, for otherwise there will be two hosts with the same address. Worse, if the initial server attempts to continue serving the connection, confusion may ensue at the client's transport layer.

Current approaches take advantage of the first requirement to ameliorate the second. Since both the initial server and the failover server must be co-located, so-called layer-four switches or "Web switches" are placed in front of server groups. Web switches multiplex incoming requests across the servers, and rewrite addresses as packets pass through. This enables multiple servers to appear to the external network as one machine, providing client transparency. The obvious drawback of this approach, however, is that all servers share fate with the switch, which may become a performance bottleneck.

By using explicit connection migration, which exposes the change in server to the client, we remove both of these restrictions. (Indeed, our approach further empowers the client to take part in the selection of the new server.) Servers can now be replicated across the wide-area, and there is no requirement for a redirecting device on the path between client and server.[5]

Explicitly informing the client of the change in server provides robust behavior in the face of server resurrection as well. If the previous server attempts any further unsolicited communication with the client—possibly due to network healing, server restart, or even a false death report by the health monitoring system, the client simply rejects the communication in the same fashion as any other unsolicited network packet.

Our architecture leverages the absence of the co-location requirement by allowing the client to select its candidate server of choice. This can be done in the transport layer by simply accepting the first migration message to arrive. Assuming that all servers in the support group are notified of the current server's failure (section 2.1), the first request to arrive at the client is likely from the server best equipped to handle the message. The response time of a candidate server is the sum of the delay at the server and propagation delay of the request to the client. While not guaranteed to be the case due potentially unstable network conditions in the Internet, in general a candidate server with good (low latency) connectivity and low load is likely to win out over a loaded candidate server with a poor route to the client.

[5]The stream mapper is not such a device; rather, it is a software module that allows a server application to participate in the soft-state synchronization protocol and connection resumption mechanism.

Clearly there are exceptions to this rule, but we believe that it is a reasonable starting point. We note that if a more sophisticated decision process is desired it can be implemented either at the candidate servers, since each is aware of not only the client to be contacted, but has a reasonable knowledge of the identities of the other members of the support group, or the client application, or both.

3.3.1 Protocol requirements for migration

Our architecture fundamentally requires the ability to migrate transport-level connections. While not widely available today, we believe that this capability is a powerful way to support both load-balancing and host mobility, and can be deployed incrementally by backward-compatible extensions to protocols like TCP [22] and as an inherent feature of new protocols like the Stream Control Transport Protocol (SCTP) [25].

In addition to basic address-change negotiation, a migrateable transport protocol must provide two features to support our failover architecture: (i) it should be possible for the migration to be securely initiated by a different end-point from the ones used to establish the connection, and (ii) the transport mechanism must provide a method for extracting the sequence information of the last successfully received data at the receiver, so that the resumption can proceed correctly. Furthermore, the performance of our scheme is enhanced by the ability of multiple candidate servers to simultaneously attempt to migrate the connection, with the guarantee that at most one of them will succeed. We now address these issues in the context of TCP and SCTP.

3.3.2 TCP migration

While not a standard feature of TCP, extensions have been proposed to support connection migration, including the recently-proposed TCP Migrate Options [22, 23]. Using these options, correspondent hosts can preserve TCP connections across address changes by establishing a shared, local connection token for each connection. Either peer can negotiate migration to a new address by sending a special Migrate SYN segment containing the token of the previous connection from the new address. A migrating host does not need to know the IP addresses of its new attachment point(s) in advance.

Each migration request includes a sequence number, and any requests with duplicate sequence numbers are ignored (actually, they are explicitly rejected through a RST segment). This provides our scheme with the needed at-most-once semantics—each candidate server sends a Migrate Request with the same sequence number; the first packet to be received at the client "wins," and the rest are rejected. Furthermore, once the connection is migrated, packets from the previous address are similarly rejected, hence any attempt by the previous server to service the client (perhaps due to network healing or a erroneous death report) are actively denied.

The Migrate SYN as originally specified used the sequence number of the last transmitted data segment. When packets are lost immediately preceding migration, retransmissions from the new address carry a sequence number earlier than the Migrate SYN. Unfortunately, several currently-deployed stateful firewalls block these seemingly spurious data segments, considering them to be a security risk.

We remedy the situation by modifying the semantics of the Migrate SYN to include the sequence number of the last data segment successfully acknowledged by the client. This ensures that all data segments transmitted from the new address will be sequenced after the Migrate SYN. Further, since a host cannot reliably know what the last successfully acknowledged segment number is (since ACKs may be lost), we relax the enforcement of sequence number checking on Migrate SYNs.

By allowing the Migrate SYN to fall outside of the current sequence space window, however, an attacker does not need to know the current sequence space of a connection to hijack it. Hence, in our extended model, only the secured variant of the Migrate Options provides protection against hijacking by an eavesdropper. The secured form of the Migrate Options uses an Elliptic Curve Diffie-Hellman key exchange during the initial three-way handshake to negotiate a cryptographically-secure connection key. Secure Migrate SYNs must be cryptographically authenticated using this key, hence an attacker without knowledge of the connection key cannot hijack a connection regardless of the current sequence space.

This relaxation allows any server with sufficient knowledge of the initial transport state (namely the initial sequence number, connection token, and key) to request a migration. When a server with stale transport layer state assumes control of a connection, however, the client needs to flush its SACK blocks (and corresponding out-of-order packets) for proper operation of the TCP stack at the new server.[6] Otherwise, the transmission of a data segment that fills a gap in previously-transmitted out-of-order segments will cause the client to acknowledge receipt of data that the new server has not yet sent, confounding the server's TCP (and an untold number of middle boxes).

In general, a host cannot deduce the difference between a Migrate SYN issued by the original host (from a new address) that simply failed to receive some number of ACKs and a new end point. We therefore reserve one bit from the Migrate SYN to flag when a Migrate Request is coming from a host other than the one that initiated the connection [23]. This allows TCP connections migrated by the same host to avoid the negative performance implications of discarding out-of-order segments while providing correct operation in the face of end host changeover.

3.3.3 SCTP migration

Recent work in IP telephony signaling has motivated the development of a new transport protocol by the transport area of the IETF (Internet Engineering Task Force). A proposed standard, the Stream Control Transport Protocol (SCTP), provides advantages over TCP for connections to multi-homed hosts [25]. By advertising a set of IP addresses during connection establishment, a multi-homed SCTP connection supports transmitting and receiving data on multiple interfaces. We can leverage this capability to support connection migration between different servers.

Unlike the TCP Migrate options, however, all addresses to be used for an SCTP connection must be specified at connection establishment. While limiting the level of dynamism in the server pool, it still supports failover between servers that were known to the initial server at the time of connection establishment. A recent Internet Draft [26] attempts to address this issue by allowing for the dynamic addition and deletion of IP addresses from the association, although it requires the operations to be initiated by an end point already within the association. Additionally, SCTP was designed to support multi-homed hosts, as opposed to address changes, hence it does not support the at-most-once semantics required to allow multiple servers to simultaneously attempt to migrate the connection. We believe this is a deficiency in SCTP that can be addressed simultaneously while adding support for dynamic changes to the address associations.

The additional complexity of SCTP requires servers to communicate more transport-level state information, but need not increase the frequency of communication. SCTP end-points are required to emit SACK packets upon receipt of duplicate data segments, hence the new server could send a data segment with a Transport Sequence Number (TSN) that is known to be stale, eliciting a SACK with the current sequence state.

[6] This behavior is in full compliance with the SACK specification [15].

3.4 Limitations

Our architecture depends on the ability to perform the stream mapping function for objects being requested. Due to variability in header lengths, this requires access to the transport layer immediately below the application. In many cases, there is only one (non-trivial) transport protocol in use, such as TCP or SCTP. In some instances, however, transport protocols may be layered on top of each other, such as RTP over TCP. In this instance, both migration and stream mapping must be performed at the highest level, namely RTP.

Independent of the transport- and network-layer issues handled by our architecture, particular applications may attempt to enforce semantics that are violated by a server change; clearly such applications cannot be handled in a transparent fashion. In particular, the object being served may have changed since the connection was initially opened, resulting in indeterminate behavior if the application isn't aware of this. Further, some applications make decisions at connection establishment based on server- or time-sensitive state, and do not normally continue to reevaluate these conditions. For instance, current applications may make authorization decisions based on the IP address of its original peer, or an HTTP cookie or client certificate that has since expired. Such applications can be notified of the connection migration by the stream mapper, however, and allowed to perform any required steps to resynchronize themselves before resuming transmission.

Despite these limitations, the common case of a long-running uni-directional download from a static file or consistent stream is handled by our architecture in an application-transparent manner. As described in the following sections, our implementation allows a connection to be migrated at any point after the initial object request completes. Furthermore, our scheme can survive cascaded failures (when the back-up server fails before completing the connection migration) due to the semantics of our state distribution mechanism and the robustness of the transport-layer migration functionality.

4 Implementation

Our current prototype implementation supports TCP applications using the Migrate options [22, 23]. The soft-state distribution mechanism and stream mapping functionality are implemented as a *wedge* that runs on the server and proxies connections for the local content server. Our current implementation uses Apache 1.3, and is compatible with any Web server software that supports HTTP/1.1 range requests. It is possible to optimize performance by handing off the TCP relaying to the kernel after the request parameters have been determined, in a manner akin to MSOCKS [14].

4.1 The wedge

The wedge is a TCP relay that accepts inbound connections from clients, and forwards them to the local content server. It listens in on the initial portion of a connection to identify the object being requested by the client, examines the returned object to determine the parameters necessary to resume the connection if necessary, and then hands the relaying off to a generic TCP *splice* to pass the data between the client and the server. The use of a user-level splice to copy data from one TCP connection (server-to-wedge) to another (wedge-to-client) results in some performance degradation compared to an in-server or in-kernel implementation, but permits a simple and clean implementation that can be used with a variety of back-end servers, not just one modified server.

Each application protocol handled by the wedge requires a parsing module to identify the requested object and strip out any protocol headers on the resumed connection. The architecture of the wedge in the context of HTTP is shown in figure 2.

The wedge first passes the connection through the HTTP GET parser, which watches the connection for a GET request to identify the requested object. The parser passes the request along to the back-end server, and hands off further control of the connection to the HTTP header parser. The parser counts the length of the HTTP headers to identify the offset in the data stream of the beginning of the object data, and forwards the headers back to the client. It then passes control to the generic data relay, which maintains control of the connection until termination.

Finally, a protocol-independent soft-state distributor periodically examines the set of active connections and sends information about each connection (over UDP) to its support group. This information includes:

- Client IP address and port number.
- Takeover sequence number.
- TCP Migrate fields (connection token, key, etc.).
- Application-specific object parameters, including the name of the object.

Each server maintains a table of all the connections in its support group(s). When a server receives notification that a peer has failed, it determines if it is in the support group for any connections previously managed by the dead server (using the hashing mechanism described in section 3.1), and attempts to resume the connections.

To do so, it migrates the client connection (section 4.2),

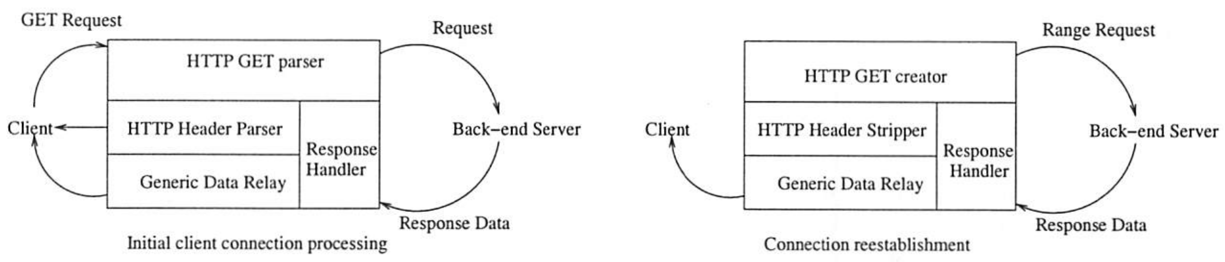


Figure 2: The wedge handling a new connection (left) and taking over an existing connection (right).

computes the new offset into the data stream by comparing the current TCP sequence number to the connection's initial sequence number, and then passes the client connection to the protocol re-establishment module to continue the connection. For HTTP, the re-establishment module sends an HTTP range request to the local server, strips out the protocol headers, and then relays the data to the client, seamlessly resuming the transfer.

To avoid race conditions, connection migrations are sequenced. If a server hears an announcement for a connection it believes it is currently managing, it checks the sequence number. If that number is greater than its own, it can safely infer that the connection has been migrated elsewhere (due to a load-balancing policy decision or an erroneous death report), and it may terminate its connection and stop advertising it to others. Similarly, peers accept only the most recent announcement for a connection, facilitating convergence after a migration.

4.1.1 Soft-state information dissemination

Soft-state information updates are sent every `UPDATE_SEC` seconds, and expired from remote servers after `CACHE_TIMEOUT` seconds. The proper values for these parameters depend on several factors [19]: connection lengths, packet loss rates between servers, and connection frequency. Our default values for these are an update frequency of three seconds and a timeout period of 10 seconds, which result in acceptable update frequencies when serving typical streaming media connections, without undue state dissemination overhead. The optimal values for these figures would, of course, vary from site to site.

4.1.2 Persistent connections

Our wedge implementation does not currently support persistent connections, but could do so with only minor modification. The wedge would need to continually monitor the client side of the connection (instead of blindly handing the connection off to a splice) and watch for further object requests. When it receives a new request, the wedge would begin announcing it via the soft-state distribution protocol. Using techniques from the LARD persistent connection handler [4], the splice can still perform fast relaying of the bulk data from the server to the client. An in-server implementation of the soft-state dissemination protocol would of course eliminate this need.

4.2 Socket interface

In order for a new server to handle a migrated connection, the wedge must preload a socket with sufficient transport information. Conversely, this information must be extracted from the socket at the previous server and communicated to the new server. We have implemented two new system calls, *setsockstate()* and *getsockstate()*, to provide this functionality.

The *getsockstate()* call packages up the TCP control block of an existing TCP socket, while the *setsockstate()* call injects this information into an unconnected TCP socket. Only certain parts of the control block are relevant, however, such as the sequence space information, TCP options, and any user-specified options such as an MTU clamp. Other, unportable state such as the retransmission queue, timers, and congestion window are returned by *getsockstate()*, but not installed into the new socket by *setsockstate()*. Invoking the *connect()* system call will then cause the socket to attempt to migrate the connection (as specified by the preloaded state) to the new socket.

After migration is completed (the *connect()* call succeeds), the new server can compare the current sequence number to the initial sequence number returned by *getsockstate()* to determine how many bytes have been sent on the connection since its establishment.

4.3 Migration

Figure 3 presents a `tcpdump` trace of a failover event, collected at the client. There are four hosts in this example: the client, cl, and three servers, sA, sB, and sC. To simulate a diverse set of realistic network conditions, the servers are routed over distinct DummyNet [21] pipes with round trip times of approximately 10ms, 40ms, and 200ms respectively. Each pipe has a bottleneck bandwidth of 128Kb per second. Initially the client is retrieving an object from sA. After some period of time, an

Initial Data Transmission:

```
 0.00000 cl.1065 > sA.8080:  .  ack 0505 win 31856
```

——— (Erroneous) *sA* Death Pronouncement Issued ———

```
 0.08014 sA.8080 > cl.1065:  P 0505:1953(1448) ack 1 win 31856
```

Successful Connection Migration to sB:

```
 0.09515 sB.1033 > cl.1065:  S 0:0(0) win 0 <migrate PRELOAD
1>
 0.09583 cl.1065 > sB.1033:  S 0:0(0) ack 1953 win 32120
 0.14244 sB.1033 > cl.1065:  .  ack 1 win 32120
```

Continued Data Transmission from sA:

```
 0.17370 sA.8080 > cl.1065:  P 0505:1953(1448) ack 1 win 31856
 0.17376 cl.1065 > sA.8080:  R 1:1(0) win 0
```

Failed Connection Migration Attempt by sC:

```
 0.17423 sC.1499 > cl.1065:  S 0:0(0) win 0 <migrate PRELOAD
1>
 0.17450 cl.1065 > sC.1499:  R 0:0(0) ack 1 win 0
```

Resumed Data Transmission from sB:

```
 0.24073 sB.1033 > cl.1065:  P 1953:3413(1460) ack 1 win 32120
 0.25663 cl.1065 > sB.1033:  .  ack 3413 win 31856
 0.33430 sB.1033 > cl.1065:  P 3413:4873(1460) ack 1 win 32120
 0.42776 sB.1033 > cl.1065:  P 4873:6333(1460) ack 1 win 32120
 0.42784 cl.1065 > sB.1033:  .  ack 6333 win 31856
 .
 .
 .
```

Figure 3: An annotated failover trace (collected at the client) depicting the migration of a connection to one of two candidate servers.

erroneous death pronouncement is simultaneously delivered to the servers by a simulated health monitor that declares sA dead. This pronouncement is received by the servers at approximately 0.073s (not shown).

As figure 3 shows, each of the other servers in the support group immediately attempts to migrate the connection. Due to their disparate path latencies, the Migrate SYNs arrive at different times. With a path latency of only 20ms, the Migrate SYN from sB arrives first, and is accepted by the client.

The next section of the trace shows the robustness of our scheme in the face of mistaken death pronouncements. The previous simulated announcement by the health monitor was in error; sA is in fact still operational. Furthermore, there are several outstanding unacknowledged data segments (including the segment seen in the trace at time 0.08014), as the client does not emit any further ACKs to sA once it has migrated to sB. Hence sA times out and retransmits the most recent data segment. The client responds by sending a RST segment, informing sA it is no longer interested in receiving further transmissions. (The remaining retransmissions and corresponding RSTs are not shown for clarity.)

Continuing on, we see that the Migrate SYN from the other candidate server, sC finally arrives approximately 100ms after the death announcement. Since the Migrate Request number (1) is identical to the previously received Migrate SYN, the client rejects the request. Note that if sB had died, and this Migrate SYN was an attempt by sC to further resume the connection, the request number would have been incremented.

The final portion of the trace shows the resumed data transmission, continuing from the last contiguous received data segment (as indicated by the SYN/ACK sent by the client). Since the path from client to sB is likely different from the path to sA, the TCP congestion state is reset and the connection proceeds in slow start.

5 Performance

We conducted several experiments to study the robustness of our scheme in the presence of overly-reactive or ill-behaving health monitors, the overhead incurred, and the consequences of many connections requiring simultaneous migration.

5.1 Server stability

We first examined the performance degradation experienced by a connection as a function of the rate at which it is migrated between different servers. The lower the impact, the greater the resilience of our scheme to an imperfect health monitor or unstable load-balancing policy. In particular, we would like to isolate the highest migration frequency before performance severely degrades.

One might assume that performance degradation would increase steadily as the frequency of oscillations between servers increases. Hence we conducted a series of simple experiments where a client was connected to two servers over distinct links, each with a round-trip propagation time of 40ms and distinct bottleneck bandwidths of 128Kbps. The bottleneck queue size from both servers was 14 KBytes, substantially larger than the bandwidth-delay product of the path. All graphs in this section represent data collected at the client.

Contrary to our initial intuition, we find that the degradation is non-monotonic in the oscillation frequency for this experiment. This is shown in figure 4, which depicts the progression of five separate downloads, each subjected to a different rate of oscillation. The connection served entirely by one server performs best, but the other rates deviate in an unexpected manner.

While the traces and exact numbers we present are specific to our link parameters, they illustrate three important interactions. The first is intuitive: the longer the time between server change events, the higher the throughput, because there is less disruption. This explains the decreasing overall trend and the decreasing magnitude of the "bumps" in figure 5. The second effect is due to the window growth during slow start; if migrations occur before the link bandwidth is fully utilized, throughput decreases dramatically because the connection al-

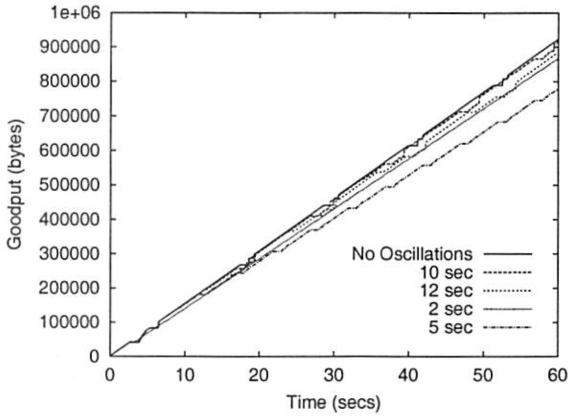


Figure 4: Connection ACK traces for varying rates of server oscillation.

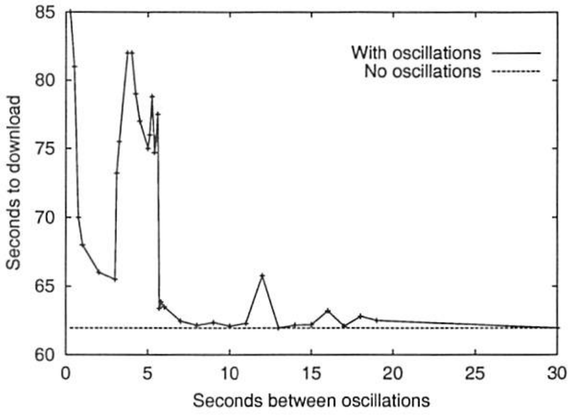


Figure 5: Download times vs. oscillation rates. A connection served entirely by one server takes 61.96 seconds to complete.

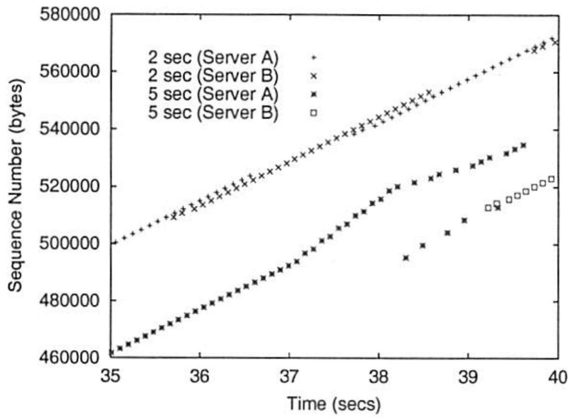


Figure 6: Sequence traces of oscillatory migration behavior.

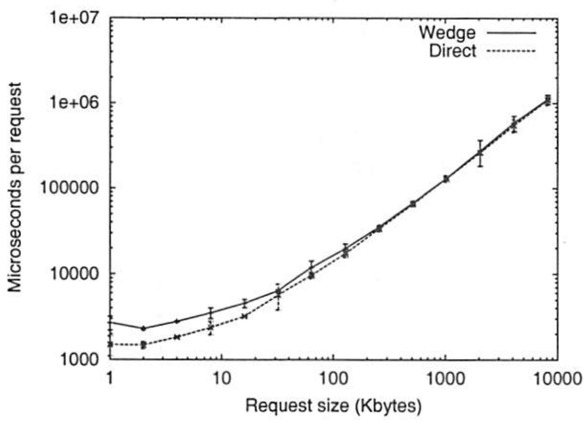


Figure 7: The request overhead of the wedge as a function of request size.

ways under-utilizes the link. This occurs at at oscillation periods less than three seconds in the figure 5. The third interaction occurs when migration occurs during TCP loss recovery, either due to slow start or congestion avoidance. In this case, the *go-back-n* retransmission policy during migration forces the connection to discard already-received data. This interaction explains the periodic "bumps" in figure 5 and is discussed in more detail below.

To illustrate the slow-start and loss recovery interactions, figure 6 examines the sequence traces for the interval from 35 to 40 seconds of the connections subjected to 2 and 5 second oscillations. At 2 seconds, connections are still ramping up their window sizes and have experienced no losses. At 5 seconds, the connections experience multiple loss events as slow start begins to overrun the buffer at the bottleneck. Four retransmissions can be observed to be successfully received, the fifth unfortunately arrives *just after* the Migrate SYN from the new server. Since the remaining data is non-contiguous, it is flushed at the receiver in accordance with the *go-back-n* policy, and retransmission resumes from substantially earlier. Similar loss interactions appear as the regular pauses in figure 4. Regardless of the exact period of interaction, the server switching overhead for realistic rates of failure is quite low.

5.2 Overhead

Micro-benchmarks of the request fulfillment time for an unloaded server are shown in figure 7. The overhead associated with wedge processing becomes negligible as request size increases to the long-running, large sessions for which our scheme is designed. The impact ranges from an additional 1ms per request (80% overhead) for 1 KByte requests to 12ms (1%) for 8 MByte requests.

The load overhead from the wedge comes from our simple TCP splicing implementation; we do not present its evaluation here. The kernel-assisted techniques mentioned in section 4 or the TCP splicing techniques de-

Method	Avg. Latency	Avg. Max Clients
Distance	34ms	2146
Optimal	38ms	1399
Backoff	67ms	1334
Round-Robin	94ms	1112
Random	94ms	1160

Table 1: Simulation distances and node with most clients for different server selection methods.

scribed by Cohen, Rangarajan, and Slye [7] can eliminate most of the TCP processing overhead, but not the connection establishment overhead; an in-server implementation can eliminate nearly all of the overhead.

5.3 Implosion

To explore the degree to which latency-dependent server selection affected server implosion, we simulated 10,000 clients served by 10 servers. (Other numbers of clients and servers had nearly identical results). The clients and servers were laid out on a random two-dimensional grid to represent the distance between them; possible latencies ranged from 0ms to 250ms.

We first tested two global methods. *Optimal* caps server occupancy at 1400 clients and performs a global minimum-latency assignment to servers. *Round-Robin* guarantees that clients are evenly distributed between (effectively random) servers. Next, we simulated three distributed methods. *Distance* uses only the distance metric to assign servers. *Random* chooses servers purely at random from the failure group. *Backoff* uses an exponential backoff based on the number of clients hosted at the server and the number of outstanding "offers."

The performace of each server selection method is shown in table 5.3. Pure distance assignment has the best latency, but suffers from severe implosion effects. Optimal does nearly as well without the implosion, but has feasibility issues in a distributed environment. Round-Robin and random assignment both result in very even distributions of clients, but have high latency. As expected, our weighted backoff method achieves a nice compromise.

6 Related work

Request redirection devices can perform per-connection load balancing, but achieve better performance and flexibility if they route requests based upon the *content* of the request. Many commercially available Web switches provide content-based request redirection [2, 6, 11, 18] by terminating the client's TCP connection at a front-end redirector. This redirector interprets the object request in a manner similar to our wedge, and then passes the request on to the appropriate server. In this architecture, however, the redirector must remain on the path for the duration of the connection, since the connection to the server must be spliced together with the client's connection.

In-line redirection inserts a potential performance bottleneck, and the benefits of TCP splicing [7] and handing off connections directly to end machines within a Web server cluster are well known. Handoff mechanisms were first explored by Hunt, Nahum, and Tracey [13], and later implemented in the LARD (Locality-Aware Request Distribution) system [17]. LARD was recently extended [4] to support request handoff between backend servers for HTTP/1.1 persistent connections [10]. A key component of the LARD implementation is a TCP handoff mechanism, which allows the front-end load balancer to hand the connection off to back-end servers after the load-balancing decision has been made. Similar functionality can now also be found in a commercial product from Resonate [20]. While all three of these mechanisms are transparent to the client, they each require the connection to be actively handed off by the front end to the back-end server. [7]

The need for previous techniques to maintain hard connection state at the front end has made developing Internet telephony systems with reliability equivalent to current circuit-switched technologies quite difficult. A new Reliable Server Pooling (Rserpool) working group has recently been formed by the IETF to examine the needs of such applications. We believe our architecture addresses many of the requirements set forth in the working group charter [16] and described in the Aggregate Server Access Protocol (ASAP) Internet Draft [24].

7 Conclusion

We described the design and implementation of a fine-grained failover architecture using transport-layer connection migration and an application-layer soft-state synchronization mechanism. Our architecture is end-to-end and transparent to client applications. It requires deployment of previously-proposed changes to only the transport protocol at the communicating peers, but leaves server applications largely unchanged. Because it does not use a front-end application- or transport-layer switch, it permits the wide-area distribution of each connection's support group.

Experimental results of our prototype Linux implementation show that the performance of our failover system is not severely affected even when connection halts and

[7] Back-end forwarding [4] allows different back ends to serve subsequent requests, but requires the previous server to forward the content (through the front end) back to the client.

resumptions occur every few seconds. Performance decreases only marginally with increasing migration frequency, with an additional contribution dependent on the loss rate of the connection immediately preceding migration. We therefore believe that this approach to failover is an attractive way to build robust systems for delivering long-running Internet streams.

The techniques described in this paper are applicable to a variety of systems, in addition to the traditional end-to-end browser-to-server model. Although our architecture does not require application- or transport-layer switches for routing and health monitoring, it does not preclude them. For example, some commercial products (e.g., Cisco LocalDirector [6]) suggest using two co-located Web switches for redundancy. When used across long-running connections, our solution allows existing connections to be seamlessly migrated between multiple Web switches without adversely affecting performance.

Our source code is available (under GPL) at `http://nms.lcs.mit.edu/software/migrate/`.

References

[1] Akamai Technologies, Inc. `http://www.akamai.com`.

[2] Alteon Web Systems. Layer 7 Web Switching. `http://www.alteonwebsystems.com/products/whitepapers/layer7switching`.

[3] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proc. ACM SIGCOMM '98*, Sept. 1998.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. In *Proc. USENIX '99*, June 1999.

[5] Cisco Systems. Cisco Distributed Director. `http://www.cisco.com/warp/public/cc/pd/cxsr/dd/tech/dd_wp.htm`.

[6] Cisco Systems. Failover configuration for LocalDirector. `http://www.cisco.com/warp/public/cc/pd/cxsr/400/tech/locdf_wp.htm`.

[7] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proc. USITS '99*, Oct. 1999.

[8] Digital Island, Inc. Digital Island, Inc. Home Page. `http://www.digitalisland.net`.

[9] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proc. IEEE Infocom '98*, Mar. 1998.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. HyperText Transfer Protocol—HTTP/1.1. RFC 2068, IETF, Jan. 1997.

[11] Foundry Networks. ServerIron Internet Traffic Management Switches. `http://www.foundrynet.com/PDFs/ServerIron3_00.pdf`.

[12] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-based scalable network services. In *Proc. ACM SOSP '97*, Oct. 1997.

[13] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[14] D. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proc. IEEE Infocom '98*, Mar. 1998.

[15] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, IETF, Oct. 1996.

[16] L. Ong and M. Stillman. Reliable Server Pooling. Working group charter, IETF, Dec. 2000. `http://www.ietf.org/html.charters/rserpool-charter.html`.

[17] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proc. ASPLOS '98*, Oct. 1998.

[18] Radware. Web Server Director. `http://www.radware.com/archive/pdfs/products/WSD.pdf`.

[19] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proc. ACM SIGCOMM '99*, Sept. 1999.

[20] Resonate. Central Dispatch 3.0 - White Paper. `http://www.resonate.com/products/pdf/WP_CD3.0___final.doc.pdf`.

[21] L. Rizzo. Dummynet and forward error correction. In *Proc. Freenix '98*, June 1998.

[22] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. ACM/IEEE Mobicom '00*, pages 155–166, Aug. 2000.

[23] A. C. Snoeren and H. Balakrishnan. TCP Connection Migration. Internet Draft, IETF, Nov. 2000. `draft-snoeren-tcp-migrate-00.txt` (work in progress).

[24] R. R. Stewart and Q. Xie. Aggregate Server Access Protocol (ASAP). Internet Draft, IETF, Nov. 2000. `draft-xie-rserpool-asap-01.txt` (work in progress).

[25] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, IETF, Oct. 2000.

[26] R. R. Stewart, Q. Xie, M. Tuexen, and I. Rytina. SCTP Dynamic Addition of IP addresses. Internet Draft, IETF, Nov. 2000. `draft-stewart-addip-sctp-sigran-01.txt` (work in progress).

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association:

Addison-Wesley
Kit Cosper
Earthlink Network
Edgix
Interhack Corporation
Interliant
Linux Security, Inc.
Lucent Technologies
Microsoft Research
Motorola Australia Software Centre
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Raytheon Company
Sams Publishing
Sendmail, Inc.
Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
Taos: The Sys Admin Company
UUNET Technologies, Inc.

Supporting Members of SAGE:

Certainty Solutions
Collective Technologies
Electric Lightwave, Inc.
ESM Services, Inc.
Linux Security, Inc.
Mentor Graphics Corp.
Microsoft Research
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Raytheon Company
Remedy Corporation
RIPE NCC
Sams Publishing
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org*.

ISBN 1-880446-14-6